Programare C#

Bibliografie

•Herbert Schildt, *C#: A Beginner's Guide, (2001)*;

•Herbert Schildt, *C#*, Ed.Teora (traducere, 2002);

•Karli Watson et al., *Beginning Visual C#*, Wrox Press Ltd. (2002);

•Karli Watson, *Beginning C# 2005 Databases*, Wiley Publishing, Inc. (2006);

•Bradley L. Jones, *SAMS Teach Yourself the C# Language in 21 Days*, *(*2004);

•Philip Syme si Peter Aitken, *SAMS Teach Yourself the C# Web Programming in 21 Days*, (2002);

•Kris Jamsa si Lars Klander, *Totul despre C si C++ Manualul fundamental de programare in C si C++, Ed. Teora*, (traducere 2007);

Introducere

- Scurt istoric;
- Relatia dintre C# si arhitectura .NET;
- Principiile programarii orientate obiect;
- Crearea, compilarea si executia programelor C#. Exemple;

Scurt istoric

Lansat publicului in iunie 2000 si oficial in primavara anului 2002, C# este un limbaj de programare care combina facilitati testate de-a lungul timpului cu inovatii de ultim moment. Creatorii acestui limbaj au fost o echipa de la firma Microsoft condusa de Anders Hejlsberg. Desi limbajul este creat de Microsoft, acesta nu este destinat doar platformelor Microsoft. Compilatoare C# exista si pentru alte sisteme precum Linux sau Macintosh. Creat ca instrument de dezvoltare pentru arhitectura .NET, limbajul ofera o modalitate facila si eficienta de a scrie programe pentru sistemul Windows, internet, componente software etc.

C# deriva din doua dintre cele mai de succes limbaje de programare: C si C++. De asemenea, limbajul este o “ruda” apropiata a limbajului Java. Pentru o mai buna intelegere a limbajului C# este interesant de remarcat care este natura relatiilor acestuia cu celelalte trei limbaje mentionate mai sus. Pentru aceasta, vom plasa mai intai limbajul C# in contextul istoric determinat de cele trei limbaje.

Limbajul C. Programarea structurata

Limbajul C a fost inventat de catre Dennis Ritchie in anii ‘70 pe un calculator pe care rula sistemul de operare UNIX. Limbajul C s-a dezvoltat in urma revolutiei programarii structurate din anii ’60. Inainte de programarea structurata, programele erau greu de scris si de inteles din cauza logicii. O masa incalcita de salturi, apelui si reveniri, greu de urmarit era cunoscuta sub numele de cod spaghetti. Datorita sintaxei sale concise si usor de utilizat, in anii ‘80, limbajul C a devenit cel mai raspandit limbaj structurat.

Limbajul C are insa limitele sale. Una dintre acestea o reprezinta incapacitatea de a lucra cu programe mari. Limbajul C ridica o bariera atunci cand programul atinge o anumita dimensiune. Acest prag depinde de program, instrumentele folosite, programator, dar este posibil sa se situeze in jurul a 5000 de linii de cod.

Limbajul C++. Programarea orientata obiect

La sfarsitul anilor ’70 dimensiunile multor programe erau aproape de limitele impuse de limbajul C. Pentru a rezolva problema a aparut o modalitate noua de programare si anume programarea orientata obiect (POO). Limbajul C nu permitea programarea orientata obiect. Fiind cel mai raspandit limbaj, s-a dorit extinderea sa in vederea implementarii noii modalitati de programare: programarea POO.

Limbajul C++ a fost creat de catre Bjarne Stroustrup incepand din 1979, la laboratoarele Bell din Murray Hill, New Jersey. Limbajul a fost denumit initial C cu clase, iar in 1983 numele acestuia a fost modificat in C++. In esenta, C++ reprezinta versiunea orientata obiect a limbajului C. In anii ’80, limbajul C++ a suferit dezvoltarii si perfectionari masive, astfel ca in anii ’90 a devenit cel mai raspandit limbaj de programare.

Limbajul Java. Problema portabilitatii

Lucrul la acest limbaj a fost demarat in 1991 la firma Sun Microsystems. Java este un limbaj structurat si orientat pe obiecte, cu o sintaxa si filozofie derivate din C++. Aspectele novatoare se refera mai mult la modificarile mediului de programare. Aspectul esential in Java este posibilitatea de a crea cod portabil pe platforme diferite. Inainte de explozia Internetului, majoritatea programelor erau compilate si destinate utilizarii pe un anumit procesor si sub un anumit sistem de operare. Programele scrise in C si C++ se compilau intodeauna pana la cod masina executabil. Codul masina este legat de un anumit procesor si de un anumit sistem de operare. Dupa aparitia Internetului insa, la care sunt conectate sisteme cu procesoare si sisteme de operare diferite, problema portabilitatii a devenit foarte importanta.

Java a realizat portabilitatea prin transformarea codului sursa al programului intr-un cod intermediar numit *bytecode.* Acest format intermediar este executat apoi de asa numita Masina Virtuala Java (MVJ). Asadar, programele Java pot rula in orice mediu in care este disponibila o MVJ. Deoarece MVJ este usor de implementat, aceasta a fost imediat disponibila pentru un numar mare de medii.

Limbajul C#

Desi Java a rezolvat cu succes problema portabilitatii, exista unele aspecte care ii lipsesc. Una dintre acestea este *interoperabilitatea limbajelor diferite*, sau *programarea in limbaj mixt* (posibilitatea codului scris intr-un limbaj de a lucra in mod natural cu codul scris in alt limbaj). Interoperabilitatea limbajelor diferite este esentiala la realizarea sistemelor software de dimensiuni mari.

Ca parte a ansamblului strategiei .NET, dezvoltata de Microsoft, la finele anilor ’90 a fost creat limbajul C#. C# este direct inrudit cu C, C++ si Java. “Bunicul” limbajului C# este C-ul. De la C, C# mosteneste sintaxa, multe din cuvintele cheie si operatorii. De asemenea, C# construieste peste modelul de obiecte definit in in C++. Relatia dintre C# si Java este mai complicata. Java deriva la randul sau din C si C++. Ca si Java, C# a fost proiectat pentru a produce cod portabil. Limbajul C# nu deriva din Java. Intre C# si Java exista o relatie similara celei dintre “veri”, ele deriva din acelasi stramos, dar deosebindu-se prin multe caracteristici importante.

Limbajul C# contine mai multe facilitati novatoare, dintre care cele mai importante se refera la suportul incorporat pentru componente software. C# dispune de facilitati care implementeaza direct elementele care alcatuiesc componentele software, cum ar fi proprietatile, metodele si evenimentele. Poate cea mai importanta facilitate de care dispune C# este posibilitatea de a lucra intr-un mediu cu limbaj mixt.

Relatia dintre C# si arhitectura .NET

C# are o legatura deosebita cu mediul sau de rulare, arhitectura .NET. Pe de o parte, C# a fost dezvoltat pentru crearea codului pentru arhitectura .NET, iar pe de alta parte bibliotecile utilizate de C# sunt cele ale arhitecturii .NET.

Ce este arhitectura .NET ?

Arhitectura .NET defineste un mediu de programare care permite dezvoltarea si executia aplicatiilor indiferent de platforma. Aceasta permite programarea in limbaj mixt si ofera facilitati de securitate si portabilitate a programelor. Este disponibila deocamdata pentru platformele Windows.

Legat de C#, arhitectura .NET defineste doua entitati importante si anume *biblioteca de clase .NET* si *motorul comun de programare* sau *Common Language Runtime (CLR)*.

C# nu are o biblioteca de clase proprie ci utilizeaza direct biblioteca de clase .NET. De exemplu, cand se ruleaza un program care efectueaza operatii de intrare-iesire, cum ar fi afisarea unui text pe ecran, se utilizeaza biblioteca de clase .NET.

Motorul comun de programare (CLR) se ocupa de executia programelor C#. El asigura de asemenea programarea in limbaj mixt, securitatea si portabilitatea programelor. Atunci cand este compilat un program C#, sau un program in limbaj mixt, rezultatul compilarii nu este un cod executabil. In locul acestuia, se produce un fisier care contine un tip de pseudocod numit limbaj intermediar sau pe scurt IL (Intermediate Language). Acest fisier IL poate fi copiat in orice calculator care dispune de .NET CLR. Prin intermediul unui compilator denumit JIT (Just In Time), motorul comun de pogramare transforma codul intermediar in cod executabil. Procesul de conversie decurge astfel: atunci cand un program .NET este executat, CLR activeaza compilatorul JIT. Compilatorul JIT converteste IL in cod executabil pe masura ce fiecare parte a programului este neceasra. In concluzie, orice program compilat pana in format IL poate rula in orice mediu pentru care CLR este implementat. In acest fel arhitectura .NET asigura portabilitatea.

Principiile programarii orientate obiect

Metodologiile de programare s-au modificat continuu de la aparitia calculatoarelor pentru a tine pasul cu marirea complexitatii programelor. Pentru primele calculatoare programarea se facea introducand instructiunile masina scrise in binar. Pe masura ce programele au crescut s-a inventat limbajul de asamblare, in care se puteau gestiona programe mai mari prin utilizarea unor reprezentari simbolice ale instructiunilor masina. Cum programele continuau sa creasca, s-au introdus limbaje de nivel inalt, precum FORTRAN si COBOL, iar apoi s-a inventat programarea structurata.

POO a preluat cele mai bune idei de la programarea structurata, combinanu-le cu concepte noi. A rezultat o modlitate diferita de a organiza un program. In fapt, un program poate fi organizat in doua moduri: in jurul codului (mod de lucru descris de sintagma *"codul actioneaza asupra datelor"*, valabil in cazul programarii structurate) sau in jurul datelor (abordare descrisa de sintagma *"datele controleaza accesul la cod"*, valabila in cazul programarii orientate obiect).

Toate limbajele POO au patru caracteristici comune: incapsularea, polimorfismul, mostenirea si reutilizarea.

Incapsularea

Incapsularea este un mecanism care combina codul si datele pe care le manipuleaza, mentinand integritatea acestora fata de interferenta cu lumea exterioara. Incapsularea mai este numita si realizarea de cutii negre, intrucat se ascunde functionalitatea proceselor. Cand codul si datele sunt incapsulate se creaza un obiect. In cadrul unui obiect, codul si datele pot fi publice sau private. Codul si datele private sunt accesibile doar in cadrul aceluiasi obiect, in timp ce codul si datele publice pot fi utilizate si din parti ale programului care exista in afara acelui obiect.

Unitatea fundamentala de incapsulare este clasa. Clasa specifica datele si codul care opereaza asupra datelor. O clasa defineste forma unui obiect. Sau altfel spus, o clasa reprezinta o matrita, iar un obiect reprezinta o instanta a clasei.

Polimorfismul

Polimorfismul este calitatea care permite unei interfete sa aiba acces la un grup generic de actiuni. Termenul este derivat dintr-un cuvant grecesc avand semnificatia “cu mai multe forme”. Spre exemplu, sa presupunem ca avem o nevoie de o routina care sa returneze aria unei forme geometrice, care poate fi un triunghi, cerc sau trapez. Intrucat ariile celor trei forme se calculeaza diferit, routina trebuie sa fie adaptata la datele pe care le primeste incat sa distinga despre ce fel de forma este vorba si sa returneze rezultatul corect.

Conceptul de polimorfism este exprimat prin sintagma “o singura interfata mai mlte metode”.

Mostenirea

Mostenirea este procesul prin care un obiect poate dobandi caracteristicile altui obiect. Analogia cu conceptul de animal este elocventa. Spre exemplu, sa consideram o reptila. Aceasta are toate caracteristicile unui animal, insa in plus are si o alta caracteristica, si anume: sangele rece. Sa consideram un sarpe. Acesta este o reptila lunga si subtire care nu are picioare. Sarpele are toate caracteristicile unei reptile, insa poseda si propriile sale caracteristici. Asadar, un sarpe mosteneste caracteristicile unei reptile. O reptila mosteneste caracteristicile unui animal. Asadar, mecanismul mostenirii este cel care face posibil ca un obiect sa fie o instanta a unui caz mai general.

Reutilizarea

Atunci cand este creata o clasa, aceasta poate fi utilizata pentru a crea o multime de obiecte. Prin utilizarea mostenirii si incapsularii clasa amintita poate fi reutilizata. Nu mai este nevoie sa testam codul respectiv ci doar a il utlizam corect.

Crearea, compilarea si executia programelor C#. Exemple;

Sa consideram urmatorul program C# simplu:

```
/* Acesta este un program simplu in C#
Denumiti programul: Example1.cs */
using System;
class Example1
{
    //orice program in C# contine metoda Main()
    public static void Main()
    {
        Console.WriteLine("This is my first C# program");
    }
}
```

Exista doua moduri de a edita, compila si rula un program in C#. In primul rand se poate utiliza compilatorul linie de comanda csc.exe. A doua posibilitate este de a utiliza utilizati mediul Visual Studio .NET. In primul caz trebuie parcursi urmatorii pasi: introduceti textul programului cu ajutorul unui editor de texte si salvati fisierul utilizand extensia *cs,* spre exemplu *Example1.cs;* apoi compilati programul precizand numele fisierului in linia de comanda (C:\>csc Example1.cs); in final rulati programul in linia de comanda (C:\>Example). In cel de-al doilea caz creati un nou proiect C# selectand: File|New|Project, apoi Visual C# Projects|Empty Project. Dupa ce ati creat proiectul, executati click dreapta pe fereastra Solution. Utilizand meniul aparut selectati Add apoi Add New Item | Local Project Items| C# Code File. Introduceti textul, salvati proiectul, compilati proiectul selectand Build si in fine rulati programul selectand Start Without Debugging din meniul Debug.

Programul de mai jos creaza o aplicatie Windows.

```
using System;
using System.Windows.Forms;

 public class MyForm : Form
 {
   private TextBox txtEnter;
   private Label lblDisplay;
   private Button btnOk;

   public MyForm()
   {
       this.txtEnter = new TextBox();
       this.lblDisplay = new Label();
       this.btnOk = new Button();

       this.Text = "Prima mea aplicatie
                              Windows!";
       this.Size=new System.Drawing.Size(320,
                 300);

       // txtEnter 1
        this.txtEnter.Location = new
                 System.Drawing.Point(16, 32);
       this.txtEnter.Size = new
                 System.Drawing.Size(264, 20);

       // lblDisplay
       this.lblDisplay.Location = new
                 System.Drawing.Point(16, 72);
       this.lblDisplay.Size = new
                 System.Drawing.Size(264, 128);

        // btnOk
       this.btnOk.Location = new
                 System.Drawing.Point(88, 224);
       this.btnOk.Text = "OK";
       this.btnOk.Click +=new
                 System.EventHandler(this.btnOK_Click);
       // MyForm
       this.Controls.AddRange(new Control[] {
                 this.txtEnter, this.lblDisplay, this.btnOk});
       }

       static void Main ()
       {
                 Application.Run(new MyForm());
       }

       private void btnOK_Click(object sender,
       System.EventArgs e)
       {
                 lblDisplay.Text = txtEnter.Text + "\n" +
       lblDisplay.Text;
       }
}
```

Programul de mai jos descompune un numar natural in factori primi

```
using System;

class Descompunere
{
    public static void Main()
    {
        int n;
        int count=2;
        string l;
        Console.WriteLine("Introduceti numarul natural n");
        l=Console.ReadLine();
        n = int.Parse(l);
        Console.Write("{0}=", n);
        while (count <= n)
        {
            while (n % count == 0)
            {
                n = n / count;
                Console.Write("{0} ", count);
            }
            count++;
        }

    }
}
```

Tipuri de date si operatori

- Tipuri valorice in C#
- Partile componente ale unei aplicatii C#
- Literali
- Variabile
- Operatori
- Conversia tipurilor de date
- Studiul expresiilor

Tipuri valorice in C#

Tipurile de date si operatorii stau la baza oricarui limbaj de programare. C# ofera o gama larga de tipuri de date si operatori. Vom incepe prin examinarea tipurilor de date fundamentale in C#. Inainte de aceasta, amintim ca limbajul este puternic tipizat. Aceasta inseamna ca pentru toate operatiile, compilatorul realizeaza verificari asupra compatibilitatii tipurilor.

Limbajul C# include doua categorii generale de tipuri predefinite: *tipuri valorice* si *tipuri referinta*. Tipurile referinta din C# sunt definite de clase. Studiul acestora il vom face atunci cand vom discuta despre clase. La baza limbajului C# stau 13 tipuri valorice numite si tipuri simple. Aceasta datorita faptului ca exista o relatie directa intre tipurile de date C# si tipurile de date .NET.

Din ratiuni de portablitate, in C#, fiecare dintre tipurile valorice are domeniu fix de valori. Daca de exemplu in limbajul C, o varabila de tip int este reprezentata pe 2 octeti sau 4 octeti, in functie de platforma utilizata, in C# unei variabile de tip int, calculatorul ii aloca 4 octeti, indiferent de mediul de executie.

In tabelul de mai jos sunt prezentate aceste tipuri

Tipul in C#	Tipul in .NET	Semnificatia	Largimea (in octeti)	Domeniul
bool	System.Boolean	Valorile de adevar (adevarat/fals)	1	false(0) *la* true(1)
char	System.Char	Caractere	2	0 *la* 65535
byte	System.Byte	Intregi pe 8 biti, fara semn	1	0 *la* 255 (0 la 2^{8}-1)
sbyte	System.Sbyte	Intregi pe 8 biti, cu semn	1	-128 *la* 127
short	System.Int16	Intregi in forma scurta	2	-32768 *la* 32767
ushort	System.Uint16	Intregi in forma scurta, fara semn	2	0 *la* 65535 (0 la 2^{16}-1)
int	System.Int32	Intregi	4	-2147483648 *la* 214748367
uint	System.Uint32	Intregi, fara semn	4	0 *la* 4294967295 (0 la 2^{32}-1)
long	System.Int64	Intregi in forma lunga	8	-9223327036854775808 *la* 9223372036854775807
ulong	System.Uint64	Intregi in forma lunga, fara semn	8	0 *la* 18446744073709551615 (0 la 2^{64}-1)
float	System.Single	Virgula mobila, simpla precizie	4	$1.5x10^{-45}$ *la* $3.4x10^{38}$
double	System.Double	Virgula mobila, dubla precizie	8	$5.0x10^{-324}$ *la* $1.7x10^{308}$
decimal	System.Decimal	Tip numeric cu 28 cifre semnificative	16	$1.0x10^{-28}$ *la approx.* $7.9x10^{28}$

Intregi

In C# sunt definite noua tipuri intregi: *char, byte, sbyte, short, ushort, int, uint, long* si *ulong.*

Tipul *char.* Caracterele nu sunt reprezentate pe 8 biti ca in alte limbaje (spre exemplu C sau C++). In C# se utilizeaza modelul Unicode. Acesta defineste un set de caractere care poate repezenta caracterele din toate limbile de pe Pamant. Setul de caractere ASCII pe 8 biti, cuprins intre 0 si 127 este o submultime a modelului Unicode. Putem atribui o valoare de tip caracter daca includem caracterul intre apostrofuri simple. Exemplu: *char ch='M' ;*

Desi *char* este de tip intreg, nu poate fi amestecat la intamplare cu valori intregi deoarece nu se efectueaza conversii automate intre char si celelalte tipuri intregi. Codul de mai jos este incorect: *char ch=77*; Motivul pentru care instructiunea nu functioneaza este ca 77 este o valoare intreaga si nu este convertita automat la tipul char. Pentru a converti o valoare intreaga intr-un char vom realiza o conversie explicita (un cast). Pentru corectarea codului din exemplul anterior trebuie sa rescriem codul in forma: *char ch=(char) 77*;

Celelalte tipuri intregi sunt utilizate pentru calcule numerice. Sunt definite atat versiuni cu semn cat si fara semn. La fel ca in limbajul C, diferenta intre intregii cu semn si cei fara semn este data de interpretarea bitului cel mai semnificativ. Daca este specificat un intreg cu semn, atunci numarul este pozitiv daca bitul de semn este 0 si respectiv negativ daca bitul de semn are valoarea 1. Sa consideram cateva exemple:

sbyte: 00000001 (nr. 1); 01111111 (nr. 127); 10000000 (nr. -128); 11111111 (nr. -1);

byte: 00000001 (nr. 1); 01111111 (nr. 127); 10000000 (nr. 128); 11111111 (nr. 255);

Exemple: 1. Conversie implicita. 2. Conversie Explicita

```
using System;
class Demo
{   public static void Main()
    {   char sourceVar='a';
        ushort destinationVar;
        destinationVar = sourceVar;
        Console.WriteLine("destinationVar val:{0}", destinationVar);
    }
}
```

```
using System;
class Demo
{   public static void Main()
    {   ushort sourceVar = 97;
        char destinationVar;
        destinationVar = (char)sourceVar;
        Console.WriteLine("destinationVar val:{0}", destinationVar);
      }
}
```

Tipuri in virgula mobila

Tipurile in virgula mobila se utilizeaza pentru specificarea numerelor care au parte fractionara. Aceste tipuri sunt: *float* si *double*. Tipul float poate reprezenta in mod precis pana la 7 pozitii zecimale, in timp ce tipul double reprezinta 15 sau 16 zecimale exacte. Dintre cele doua, double este cel mai intrebuintat.

Tipul decimal

Tipul decimal nu este definit in C, C++ si Java. Acest tip utilizeaza 128 biti de memorie pentru a reprezenta valori cuprinse intre 1.0×10^{-28} si 7.9×10^{28}. Este destinat calculelor monetare, putand reprezenta in mod precis 28 de pozitii zecimale. Nu exista conversii automate intre tipurile decimal si float sau decimal si double. Exemplu:

```
using System;
class FolosDecimal
{
    public static void Main()
    {   decimal sold, dobanda;
        //calculul noului sold
        sold = 10000.5m;      //literalii de tip decimal trebuie urmati de m sau M
        dobanda = 0.04m;
        sold = sold * dobanda + sold;
        Console.WriteLine("Noul sold este {0} EUR",sold);
    }
}
```

Rezultat: Noul sold este 10400.520 EUR

Tipul bool

Tipul bool reprezinta valorile de adevar true sau false. Orice variabila de tip bool va lua una dintre aceste valori. Nu este definita o regula de conversie intre tipul bool si valori intregi.

Cateva optiuni de afisare

Sa consideram instructiunea: Console.WriteLine("Valoarea lui 10/3: "+10.0/3.0). Aceasta afiseaza rezultatul: Valoarea lui 10/3: 3.33333333333. Afisarea unui numar mare de zecimale este inadecvata de cele mai multe ori. De exemplu, in calcule financiare se afiseaza doua zecimale. Pentru a controla formatarea datelor numerice, utilizam urmatoarea forma a metodei WriteLine(): Console.WriteLine("sir de formatare", arg0, arg1,…, argN); Sirul de formatare contine doua elemente: caractere afisabile si specificatori de format. Specificatorii de format au forma generala *{Nr_arg, width:fmt #}*.

Nr_arg precizeaza numarul argumenului care trebuie afisat. Latimea minima este precizata de valoarea *width* , iar formatul este specificat de *fmt.* De asemenea, simbolul # marcheaza numarul minim de pozitii numerice. *Width, fmt* si # sunt optionale. Pentru afisarea valorilor numerice se pot utiliza urmatorii formati *fmt:*

fmt	Descriere	Format default	Exemple
C or c	Currency	$xx,xxx.xx	$12,345.67
D or d	Decimal	xxxxxxx -xxxxxxx	1234567 -1234567
E or e	Expo-nential	x.xxxxxxe+xxx -x.xxxxxxe+xxx x.xxxxxxe-xxx -x.xxxxxxe-xxx	1.234567e+123 -1.234567e+123 1.234567e-123 -1.234567e-123
F or f	Punct fix	xxxxxxx.xx -xxxxxxx.xx	1234567.89 -1234567.89
N or n	Numeric	xx,xxx.xx -xx,xxx.xx	12,345.67 -12,345.67
X or x	Hexa-decimal		ff (nr. 255)
G or g	General	Se utilizeaza forma cea mai compacta	

Partile componente ale unei aplicatii C#

Un limbaj de programare este compus dintr-o serie de cuvinte cheie care au semnificatii speciale. Un program utilizeaza aceste cuvinte impreuna cu alte cuvinte aditionale si simboluri intr-o forma organizata. Un program C# include urmatoarele: *spatii, cuvinte cheie C#, literali* si *identificatori*.

Limbajul C# contine urmatoarele cuvinte cheie:

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
while				

Literali

In C# literalii desemneaza valorile fixate, reprezentate intr-un mod accesibil utilizatorului. De exemplu, nr. 35 este un literal. Literalii se mai numesc si constante. Literalii in C# pot fi de orice tip valoric. Constantele de tip caracter sunt incluse intre apostrofuri (exemple: ‘a’, ‘$’, etc.) In ceea ce priveste literalii intregi, tipul fiecarui literal este cel mai mic tip intreg care permite memorarea sa, incepand de la *int.* Un literal intreg poate fi de tip: *int, uint, long sau ulong.* Pentru specificarea unui literal *uint* se adauga un *u* sau *U.* De exemplu 123 este de tip *int* in timp ce 123u este de tip *uint.* In mod analog pentru literali de tip *long* se adauga *l* sau *L* in timp ce pentru literali de tip *ulong* se adauga *ul* sau *UL*.

Literalii in virgula mobila sunt in mod implicit de tip *double.* Daca dorim sa specificam un literal de tip float adaugam *f* sau *F* (de exemplu, 123.4f este de tip float).

Pentru specificarea unui literal de tip *decimal* se adauga sufixul *m* sau *M (ex: 1.43m)*.

Pentru specificarea unui literal hexazecimal se utilizeaza prefixul 0x (ex: 1) c=0xFF; //255 in zecimal, 2) in=0x1a; //26 in zecimal).

Secvente escape pentru caractere

Majoritatea caracterelor pot fi afisate incluzand constantele de tip caracter intre apostrofuri. Exista insa cateva caractere care ridica probleme deosebite precum ghilimelele, apostroful etc., care au semnificatii speciale. Din acest motiv, C# pune la dispozitie secvente escape, care sunt utilizate in locul caracterelor pe care le reprezinta. Secventele escape sunt: \a (alarma), \b (sterge un caracter in urma, backspace), \n (linie noua), \r (revenire la cap de rand), \t (tab orizontal), \v (tab verical), \0 (nul), \’ (apostrof), \” (ghilimele), \\ (backslash).

In C# intalnim si un alt tip de literali: tipul *string*. Un *string*, reprezinta un sir de caractere inclus intre ghilimele (ex: “acesta este un string”). Pe langa caracterele obisnuite, un literal de tip string poate contine mai multe secvente escape.

De asemenea, in C# se pot utiliza literalii *“copie la indigo”.* Un astfel de program incepe cu @, urmat de un sir de ghilimele. Se pot include astfel caractere tab, linie noua, etc fara a utiliza secvente escape. Exista o singura exceptie. Pentru a obtine ghilimelele (“), trebuie utilizate doua caractere unul dupa altul (“”).

```
using System;
class StrDemo
{
   public static void Main()
   {
      Console.WriteLine("Prima linie \nAdoua linie");
      Console.WriteLine("a \t b \t c \nd \t e\t f \t");
   }
}

using System;
class Indigo
{
   public static void Main()
   {
      Console.WriteLine(@"Acesta este un literal
copie la indigo care
ocupa trei linii");
      Console.WriteLine(@"Alt exemplu
a   b   c
d   e   f");
      Console.WriteLine(@"Putem spune:""hello! """);
         }
}
```

Variabile

O variabila reprezinta o locatie de memorie cu nume, careia ii poate fi atribita o valoare. Valoarea unei variabile poate fi modificata pe parcursul programului. Variabilele sunt declarate printr-o instructiune de forma:

tip nume_var;

unde *tip* reprezinta tipul variabilei, iar *nume_var* numele variabilei. Variabilele trebuie declalarate inainte de a fi folosite, de asemenea tipul variabilei nu poate fi modificat pe parcursul duratei sale de viata. Tipul variabilei determina operatiile permise asupra variabilei.

Dupa declararea variabilei, aceasta trebuie initializata. Initializarea unei variabile poate fi facuta printr-o instructiune de atribuire:

nume_var=val;

unde *val* reprezinta valoarea atribuita variabilei *nume_var* (Exemple: *int i=10; bool f=true; foat fn=12.4f; long o=126L*). De asemenea initialzarea poate fi facuta dinamic, utilizand orice expresie valida la momentul in care variabila este initializata.

```
using System;
class Initdinamica
{
    public static void Main()
    {
        double raza, inaltime, volum;
        double Pi=4*Math.Atan(1);
        raza = 4;
        inaltime = 2;
        volum = Pi * raza * raza * inaltime;
        Console.WriteLine("Valoarea lui pi este: {0:#.####}", Pi);
        Console.WriteLine("Volumul cilindrului este:{0:g}", volum);
    }
}
```

Domeniul de valabilitate si durata de viata

La fel ca alte limbaje de programare, C# permite declararea unei variabile in cadrul unui bloc. Un bloc incepe cu o acolada deschisa si se incheie cu o acolada inchisa. Un bloc delimiteaza un spatiu de declarare numit si *domeniu de valabilitate*. De asemenea, acesta determina si *durata de viata* a acelor variabile. O variabla declarata in cadrul unui bloc isi pierde valoarea cand blocul este folosit.

Cele mai importate domenii de valabilitate sunt acelea definite de o clasa si de o metoda.

In cadrul unui bloc variabilele pot fi declarate in orice punct, dar sunt valide numai dupa declaratie.

Un alt aspect interesant care diferentiaza limbajul C# de celelalte limbaje este urmatorul: nici o variabila din interiorul unui domeniu interior nu poate avea acelasi nume cu o variabila declarata intr-un domeniu care il contine.

```
using System;
class NestVar
{
   public static void Main()
   {
      int x = 10;
      if (x == 10)
      {
         int y = 20;
         Console.WriteLine("x si y: " + x + " repectiv " + y);
      }
      Console.WriteLine("x este: " + x);
      //y=20;
    }
}

//Programul de mai jos nu poate fi compilat
using System;
class NestVar
{
   public static void Main()
   {
      int i, j;
      j=10;
      for (i = 0; i < 1; i++)
      {
         int j=1;
         j = j + 1;
         Console.WriteLine("variabila din interiorul blocului
      este: {0}", j);
      }
         Console.WriteLine("variabila din exteriorul blocului
      este: {0}", j);
    }
}
```

Exemplu: Programul de mai jos nu poate fi compilat

```
using System;
class Demo
{
    public static void Main()
    {
        int i;
            for (int j=1; j<10; j++)
            {
                i=j;
                Console.WriteLine("Valoarea variabilei i este: {0}", i);
            }
            Console.WriteLine("Ultima valoare a variabilei i este: {0}", i);
    }
}
```

Numele unei variabile

Numele unei variabile trebuie sa satisfaca urmatoarele reguli:

- primul caracter al unei variabile trebuie sa fie o litera, caracterul underscore "_" sau simbolul *at* "@" ;

-urmatoarele caractere pot fi litere, numere sau caractere underscore;

-nu pot fi utilizate cuvinte cheie drept identificatori.

De-a lungul timpului au fost utilizate diverese conventii pentru numirea variabilelor.

In prezent, platforma .NET utilizeaza urmatoarele conventii: *PascalCase* si *camelCase,* unde *Case* ar trebui sa explice in ce scop este utilizata variabila. Ambele conventii specifica utilizarea unor cuvinte multiple, scrise cu litere mici cu exceptia primei litere care este mare. Pentru camelCase exista o regula suplimentara, si anume primul cuvant este scris cu litera mica.

Exemple:

camelCase: age, firstName, placeOfBirth

PascalCase: Age, FirstName, PlaceOfBirth

Pentru variabilele simple se utilizeaza *camelCase*.

Pentru spatii de nume, clase, metode se utilizeaza *PascalCase*.

Operatori

Un operator reprezinta un simbol care determina compilatorul sa realizeze o anumita operatie matematica sau logica. Limbajul C# ofera patru categorii de operatori: *aritmetici, pe biti, relationali si logici.*

Operatori aritmetici

Limbajul C# defineste urmatorii operatori aritmetici: + (adunare), - (scadere), * (inmultire), / (impartire), % (rest sau operatorul modulo), ++ (incrementare), --(decrementare).

Atunci cand se aplica operatorul / asupra unor intregi, restul este trunchiat, de exemplu: 14/3 este egal cu 4.

O diferenta fata de C si C++ este faptul ca operatorul modulo se poate aplica atat tipurilor intregi cat si celor in virgula mobila (Ex: 14.0 % 3.0=2,

14.2 %3.0=2.2, 14.0%3.1=1.6.

Operatorii unari de incrementare si decrementare functioneaza la fel ca in C. Acestia sunt: x++ (forma postfixata), ++x (forma prefixata), x-- (forma postfixata), --x (forma prefixata)

```
using System;
class Moddemo
{
    public static void Main()
    {
        int cat, rest;
        double doubleCat, doubleRest;
        cat = 14 / 3;
        rest = 14 % 3;
        doubleCat = 14.0 / 3.0;
        doubleRest = 14.0 % 3.0;
        Console.WriteLine("14=3*{0}+{1}", cat, rest);
        Console.WriteLine("14/3={0}",doubleCat);
    Console.WriteLine("14.0 % 3.0={0}", doubleRest);
    }
}
X=10; y=++X;      //y=11
X=10; y=X++;     //y=10
```

```
using System;
class Moddemo
{    public static void Main()
   {   double doubleCat;
       doubleCat = 14 / 3;
       Console.WriteLine("14/3={0:.##}", doubleCat);
   }
}
```

Operatori relationali si logici

Operatorii relationali se refera la relatiile de ordine care pot exista intre doua valori, iar operatorii logici desemneaza modalitatile in care se pot asocia valorile de adevar true si false. Operatorii relationali se utilizeaza deseori cu cei logici. Rezultatele intoarse de operatorii relationali si logici sunt de tip bool.

Operatorii relationali sunt urmatorii: == (egal cu), != (diferit de), > (mai mare decat), < (mai mic decat), >= (mai mare sau egal cu), <= (mai mic sau egal cu).

Operatorii logici sunt: & (si), | (sau), ^ (sau exclusiv), || (sau scurtcircuitat), && (si scurtcircuitat), ! (non).

In cazul operatorilor relationali <,>,<=,>= operanzii trebuie sa apartina tipurilor pe care este definita o relatie de ordine.

In cazul operatorilor logici, operanzii trebuie sa fie de tip bool. In tabelul de mai jos sunt precizate rezultatele opearatiilor logice. S-a folosit conventia: 0=false si 1=true.

p	q	p&q	p|q	p^q	!p
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

Singura diferenta dintre operatorii & si | si respectiv operatorii scurtcircuitati && si || este faptul ca operatorii obisnuiti evalueaza intodeauna ambii operanzi in timp ce vaiantele scurtcircuitate evalueaza al doilea operand doar daca este necesar.

Exemplu:

```
using System;
class Scurtcirc
{
    public static void Main()
    {
        int n, d;
        n = 10;
        d = 2;
        if ((d !=0) && ((n%d)==0))
        Console.WriteLine(d +" este divizor a lui " +n);
        d = 0;
        if ((d != 0) && ((n % d) == 0))
         Console.WriteLine(d + " este divizor a lui " + n);
    }
}
```

Operatori pe biti

Operatorii pe biti realizeaza operatii asupra asupra unuia sau mai multor biti dintr-o valoare. Operatorii pe biti sunt urmatorii: & (si pe biti), | (sau pe biti), ^ (sau exclusiv pe biti), ~ (complementare pe biti), >>(deplasare pe biti la dreapta), <<(deplasare pe biti la stanga.

Exemple:

3|4=7; 5&7=5, 4^4=0, ~2=-3, 10>>1=5, 10<<1=20.

Precedenta operatorilor

1 Primary operators — () . [ ] ++ x -- x
2 Unary — + - ! ~
3 Multiplicative — * / %
4 Additive — + -
5 Shift — << >>
6 Relational — < > <= >=
7 Equality — == !=
8 Logical AND — &
9 Logical XOR — ^
10 Logical OR — |
11 Conditional AND — &&
12 Conditional OR — ||
13 Conditional — ?:
14 Assignment — =   *=   /=   % =   +=   -=   <<= >>= &= ^= |=
15 Increment and decrement — x++   x--

Conversia tipurilor de date

O practica frecventa in programare o reprezinta atribuirea valorii unei variabile de un anumit tip unei alte variabile avand tip diferit.

Exemplu: *int i=10; float f; f=i;*

Conversii implicite

Daca in atribuire sunt implicate tipuri compatibile atunci valoarea din partea dreapta este convertita automat la tipul din partea stanga.

Atunci cand un tip de date este atribuit unui alt tip de variabila, se efectueaza o conversie implicita (automata) de tip daca: cele doua tipuri sunt compatibile si tipul destinatie este mai cuprinzator decat tipul sursa. Daca cele doua conditii sunt indeplinite atunci are loc o conversie prin largire. (Ex: long in double se face prin conversie automata, in schimb double in long nu se poate realiza automat).

De asemenea nu exista conversii automate intre *decimal si double* sau *decimal si float* si nici intre tipurile numerice si *bool.*

Conversii explicite

In multe situatii (daca exista vreo legatura intre tipul sursa si tipul destinatie) se pot realiza conversii explicite de tip, utilizand un cast. Un cast este o directiva catre compilator de a converti un anumit tip in altul. Forma generala este: *(tip_tinta) expr;*

Exemple: double x,y; int z;… z= (int) (x/y);

int i=230; byte b; ... b=(byte)i;

Insa de fiecare data, responsabilitatea este cea a programatorului daca rezultatul obtinut este cel convenabil. Pentru a controla rezultatul se pot utiliza comenzile:

checked (expresie) ; sau unchecked (expresie)

Exemplu: Conversii explicite

```
using System;
class Demo
{
    public static void Main()
    {
        byte destinationVar;
        int sourceVar = 257;
        destinationVar = (byte)sourceVar;
     //destinationVar = checked((byte)sourceVar);
        Console.WriteLine("sourceVar val: {0}", sourceVar);
        Console.WriteLine("destinationVar val: {0}", destinationVar);
    }
}
```

Studiul expresiilor

O expresie este o combinatie valida de literali, identificatori, operatori.

In cadrul unei expresii este posibil sa amestecam doua sau mai multe tipuri de date atat timp cat acestea sunt compatibile intre ele. Conversiile se realizeaza utilizand *regulile de de promovare* a tipurilor din C#. Iata algoritmul definit de aceste reguli pentru operatii binare:

- Daca un operand este decimal atunci celalalt este promovat la decimal (cu exceptia cazului in care este de tipul double sau float, caz in care apare o eroare)
- Altfel daca un operand este *double* atunci celalalt este promovat la *double.*
- Altfel daca un operand este *float* atunci celalalt este promovat la *float.*
- Altfel daca un operand este *ulong* atunci celalalt este promovat la *ulong,* numai daca nu este de tip sbyte, short, int sau long, caz in care apare o eroare*.*
- Altfel daca un operand este *long* atunci celalalt este promovat la *long.*
- Altfel daca un operand este *uint, iar al doilea este de tip* sbyte, short sau int, ambii sunt promovati la long*.*
- Altfel daca un operand este *uint* atunci celalalt este promovat la *uint.*
- Altfel ambii operanzi sunt promovati la *int.*

Observatie: Rezultatul unei expresii nu poate fi un tip inferior lui *int.*

Instructiuni de control

- Introducerea caracterelor de la tastatura
- Instructiuni de selectie (**if, switch**)
- Instructiuni repetitive **(for, while, do-while, foreach)**
- Instructiuni de salt (**break, continue, goto, return)**

Introducerea caracterelor de la tastatura

- Citirea de la tastatura se face cu metodele **Console.Read(); sau Console.ReadLine();**
- Prima dintre ele returneaza un intreg, care trebuie convertit apoi la tipul de date **char.**
- Cea de-a doua returneaza un **string.**
- Datele de la consola sunt directionate intr-un buffer de linie. La apasarea tastei ENTER, in stream-ul de intrare se introduce secventa de caractere carriage return (\r) si line feed (\n). Faptul ca metoda Read() citeste dintr-un buffer de linie poate crea neplaceri daca este necesara o alta operatie de introducere de date, intrucat aceste date trebuie sterse din buffer (prin citire).

Exemplu:

```
/*introduceti un caracter de la tastatura iar apoi ENTER */
using System;
class citire
{
public static void Main()
{
    char ch;
    Console.WriteLine("Apasati o tasta si apoi ENTER");
    ch=(char)Console.Read();
    Console.Read();
    Console.Read();
    Console.WriteLine("Ati tastat: "+ch);
    }
}
```

Instructiunea if

Forma generala

```
if (conditie)
        {instructiune1;}
else
        {instructiune2;}
```

- instructiune1 si instructiune2 pot fi orice fel de instructiuni: de la instructiunea vida(;),pana la blocuri complexe delimitate cu {} sau alte instructiuni. In lipsa delimitarii corespunzatoare,va apare un mesaj de eroare! Se recomanda ca si o singura instructiune sa fie delimitata de {}.
- daca expresia conditionala este adevarata se executa instructiune1, altfel (daca exista) se executa instructiune2.
- clauza *else* este optionala. In lipsa ei, daca conditia s-a evaluat cu false, se trece la urmatoarea instructiune din program.

Exemplu:

```
using System;
class Testif
{
    public static void Main()
    {
        char ch, rasp = 'A';
        Console.WriteLine("Introduceti un
      caracter");
        ch = Convert.ToChar(Console.ReadLine());
        if (ch == rasp)
        {
            Console.WriteLine("Ati nimerit
      raspunsul");
        }
        else
        {
            Console.WriteLine("Nu ati nimerit
      raspunsul");
        }
    }
}
```

Instructiunea switch

Valoarea expresiei *expresie*, care controleaza instructiunea si care poate fi de tip **char, short, byte, int** sau **string** se compara cu constantele (literalii) *exp_const*.

În caz de egalitate se execută instrucţiunea corespunzătoare**.**

Dacă valoarea determinată diferă de oricare din constantele specificate, se execută instrucţiunea specificată la **default, care apare o singură dată, nu** neaparat la sfârşit. Dacă **default lipseşte se iese din switch.**

Valorile constantelor trebuie sa fie diferite; ordinea lor nu are importanţă.

Contrar limbajelor C si C++, in C# daca un **case** este prevazut cu o serie de instructiuni atunci in mod obligatoriu acesta se termina cu **break;**

Acoladele ce grupeaza mulţimea **case-urilor sunt** obligatorii. După fiecare **case pot apare mai multe** instrucţiuni fără a fi grupate în acolade.

Instructiunea swich are forma generala:

```
switch (expresie)
{
case exp_const1:
        secventa instr1;
        break;
case exp_const2:
        secventa instr1;
        break;
case exp_constn:
        secventa instrn;
        break;
default:
        instructiune;
        break;
}
```

Instructiune switch valida pt. C si C++ insa neacceptata in C#

```
int i=2;
switch(i)
{
case 1:Console.WriteLine(“ 1”);
case 2: Console.WriteLine (“ 2”);
case 3: Console.WriteLine (“ 3”);
case 4: Console.WriteLine (“ 4”);
default: Console.WriteLine (“ etc.”);
}
```

In C si C++Se va afisa: 2 3 4 etc!

In C# genereaza eroare.

Instructiunea switch

```
int i=2;
switch(i)
{
case 1:Console.WriteLine(“ 1”);break;
case 2: Console.WriteLine (“ 2”); break;
case 3: Console.WriteLine (“ 3”); break;
case 4: Console.WriteLine (“ 4”); break;
default: Console.WriteLine (“ etc.”); break;
}
```

Se va afisa:

2

Chiar daca nu este permis ca o secventa case sa continue cu o alta, este posibil ca doua sau mai multe instructiuni case sa refere aceeasi secventa de cod. Exemplu:

```
switch (nota)
{
case 1:
case 2:
case 3:
case 4:
Console.WriteLine("Nota nesatisfacatoare.");
break;
//...
}
```

Se va obtine acelasi mesaj (Nota nesatisfacatoare) pt. oricare din valorile 1, 2, 3 sau 4.

Instructiunea while

Forma generala:

```
while (expresie)
{
        instructiuni;
}
instructiunea_urmatoare
```

Se evaluează *expresie: dacă valoarea sa este true se* execută *instructiunile* din blocul buclei şi controlul este transferat înapoi, la începutul instrucţiunii while. Dacă valoarea expresiei este *false* se execută *instructiunea_urmatoare.*

Aşadar *instructiune se execută de zero sau mai multe* ori.

Exemplu:

```
long factorial=1;
int i=1, n;
 while (i++ < n)
    {
    factorial *= i;
    }
```

Instructiunea do…while

Forma generala:

do
{
instructiune
}
while (*expresie);*
instructiunea_urmatoare

Se execută *instructiune.* Se evaluează *expresie: dacă valoarea sa este nenulă* controlul este transferat înapoi, la începutul instrucţiunii do..while; **dacă valoarea este nulă** se execută *instructiunea_urmatoare.* **Aşadar *instructiune se execută o dată sau de mai multe* ori**

```
using System;
class Testdo
{
   public static void Main()
   {
      char ch='a';
      Console.WriteLine("Introduceti un sir de caractere");
      do
      {
         ch = (char)Console.Read();
         Console.WriteLine("Ati introdus " + ch);
         Console.Read();
         Console.Read();
      }
      while (ch != 'x');
   }
}
```

Codul de mai sus citeste si afiseaza caracterele citite de la tastatura pana cand se citeste x.

Instructiunea for

Forma generala:

for (*initializare; conditie; iteratie)*

{instructiune*}

instructiunea_urmatoare

Initializarea este de regula o instructiune de atribuire care fixeaza valoarea initial a variabilei de control al buclei. *Conditia* este o expresie de tip bool care stabileste daca bucla continua ciclarea. Expresia *iteratie* stableste cantitatea cu care variabla de control al buclei se modifica la fiecare repetare a buclei.

Una, doua sau toate trei dintre expresii pot lipsi, dar cei doi separatori sunt obligatorii.

Exemple:

```
int  i, suma=0;
for(i = 1; i<=100; i++)suma+=i;

int suma, i;
for(suma = 0, i=0; i <= 100; suma += i++);

int i = 0;
char c;
for(; (c = (char)Console.Read()) != '\n'; ++i)
Console.WriteLine(c);
```

Instructiunea break

Se utilizeaza in cadrul buclelor for, while sau instructiunii switch. Produce ieşirea din bucla sau din switch şi trece controlul la instrucţiunea următoare

Exemplu:

```
for(int i=0;i<=10;i++)
    {
     if(i==7)
          break;
     Console.WriteLine(i);
    }
```

Bucla for va rula incepand de la 0. Valoarea 7 nu va mai fi afisata deoarece instructiunea if se va evalua cu true, determinand iesirea din bucla for. Se va afisa 0,1,2,..,6

Instructiunea continue

Se utilizeaza in cadrul buclelor for, while sau instructiunii switch. Intrerupe executia iteratiei curente şi trece controlul la iteratia urmatoare.

Exemplu:

```
for(int i=0;i<=10;i++)
    {
     if(i==7)
          continue;
    Console.WriteLine(i);
    }
```

De aceasta data nu se mai iese din bucla for, ci se sare la urmatoarea iteratie. Se va afisa 0,1,2,..,6,8,..,10

Instructiunea goto

Instructiunea goto muta controlul execuţiei programului către o instrucţiune etichetată (în cadrul aceleiaşi metode!). Această instrucţiune poate lua următoarele forme:

- **goto etichetă;**
- **goto case expresie_constantă;**
- **goto default;**

Folosirea acestei instructiuni NU este recomandata!

Exemplu (Utilizarea instructiunii goto)

```
using System;
class TestGOTO
{
    public static void Main()
    {
        const int a = 5;    const int b = -5;
        int x, cod_err = -1;
        x = int.Parse(Console.ReadLine());
        if (x == 0)
        {
            cod_err = 0;     goto eroare;
        }
        else if (x<b)
        {
            cod_err = 1;      goto eroare;
        }
        else if ( x>a)
        {
            cod_err = 2;       goto eroare;
        }
eroare: //eticheta unde sare
        switch (cod_err)
        {
            case 0:
                Console.WriteLine(" Eroare x ={0}!", x);
                break;
            case 1: Console.WriteLine(" Eroare: Depasire inferioara!");
                break;
            case 2:
                Console.WriteLine(" Eroare: Depasire superioara!");
                break;
            default:
                Console.WriteLine("Nr. introdus este x={0}", x);
                break;
        }
        Console.ReadKey();
    }
}
```

Functie de valoarea variabilei x, programul afiseaza un anumit mesaj.

Clase.Obiecte.Metode

CLASE SI OBIECTE -introducere

- O clasă descrie unul sau mai multe obiecte care pot fi precizate printr-un set uniform de date şi cod (funcţionalitate);
- Orice obiect are memoria sa proprie unde se păstrează valorile tuturor datelor sale;
- Clasa defineşte caracteristicile si comportarea obiectelor. Reprezinta un set de planuri care precizeaza cum poate fi construit un obiect;
- Orice obiect are un tip; un obiect este o *instanţă a unei clase*

Un obiect este caracterizat de:

- nume – un identificator;
- contine date de un anume tip si metode (servicii, operaţii) – funcţii care accesează datele obiectului.

Clase - proiectare

Definiţia unei clase presupune:

a) combinarea datelor pe care clasa le contine cu operaţiile (codul) care prelucreaza aceste date;

b) ascunderea informaţiei;

C# poate defini mai multe categorii particulare de date si membrii care contin cod, printre care: variabile, constante, proprietati, metode, constructori, destructori, indexari, enumerari, evenimente, delegari.

De regula, modul de structurare a datelor nu este cunoscut: datele sunt declarate intr-o secţiune "privată" a clasei. Accesul la date, pentru consultare, modificare, etc. se face prin intermediul metodelor clasei si proprietatilor clasei; acestea sunt declarate intr-o secţiune "publică" a clasei**.** Pot exista date si metode publice sau private, decizia este a programatorului. Datele si codul pot fi ascunse de unde si conceptul de incapsulare (incapsulare=combinare+ascundere).

Avantaje

- **Combinarea datelor:**
  - defineşte clar ce structuri de date sunt manevrate şi care sunt operaţiile legale asupra lor;
  - programul capătă modularitate;
  - scade riscul alterării datelor din exterior;
  - facilitează ascunderea informaţiei;
- **Ascunderea informaţiei:**
  - programe mai sigure şi mai fiabile;
  - eliberează clasele utilizator de grija manevrării datelor;
  - previne apariţia erorilor;
- **Încapsulare:**

  Combinare + Ascunderea informaţiei = Protejarea datelor

  - previne apariţia erorilor prin limitarea accesului la date;
  - asigură portabilitatea programelor;
  - facilitează utilizarea excepţiilor;

Clase

- O clasă încapsulează date, metode si alti membrii care acţionează asupra acelor date.
- Sintatic, o clasă se declară astfel:

```
class identificator
{
corpul clasei;
}
```

unde *identificator* reprezinta numele clasei, iar *corpul clasei* contine datele si codul din interiorul clasei. In cazul in care, corpul clasei contine numai variabile si metode atunci acesta are forma:

```
{acces tip var1;
 //…
 acces tip varN;
 acces tip-rez Metoda1(parametri)
   {corpul Metodei1;}
 //…
 acces tip-rez MetodaN(parametri)
   {corpul metodeiN}
 }
```

Exemplul 1. In exemplul de mai jos este definita o clasa si sunt create doua obiecte instanta ale acestei clase.

```
using System;
class Point
{
    public double x;
    public double y;
}
 class Segmdr
{
    public static void Main()
    {
        Point punct1 = new Point();
        Point punct2 = new Point();
        double dist;
        punct1.x = 3;
        punct1.y = 4;
        punct2.x = 5;
        punct2.y = 3;
        dist = Math.Sqrt((punct1.x - punct2.x) * (punct1.x - punct2.x) + (punct1.y - punct2.y) * (punct1.y -
       punct2.y));
       Console.WriteLine("Distanta dintre punctele ({0},{1}) si ({2},{3}) este: {4:#.##}", punct1.x,punct1.y,
       punct2.x, punct2.y, dist);
    }
}
```

Rezultat:

Distanta dintre punctele (3,4) si (5,3) este: 2,24

Observatii: -Definitia unei clase creaza un nou tip de date. In cazul de mai sus acest nou tip de date se numeste *Point.* Am utilizat acest tip de date pentru a declara doua obiecte.

-Dupa executia instructiunii Point punct1 = new Point(); punct1 va fi o instanta a clasei Point. Orice obiect de tip Point va avea copii proprii ale variabilelor instanta x si y.

-Pentru a referi variabilele instanta se utilizeaza operatorul punct (.). Acesta leaga numele unui obiect de un membru al sau. Instructiunea punct1.x = 3; atribuie variabilei x a instantei punct1 valoarea 3.

-Nu este necesara utilizarea claselor in acelasi fisier. In acest caz se compileaza programul utilizand linia de comanda:

csc numeprog1.cs numeprog2.cs …. numeprogN.cs

unde numeprogr.1cs, numeprog2.cs …. numeprogN.cs reprezinta numele fisierelor care contin clasele necesare compilarii programului. Evident doar unul dintre acestea contine metoda Main().

Metode

Definitie. Metodele sunt subroutine care prelucreaza datele definite in cadrul clasei si pot oferi accesul la acele date.

De regula interactiunea dintre o clasa si celelalte entitati dintr-un program se face prin intermediul metodelor clasei.

Forma generala a unei metode este:

```
acces tip-rez Nume(lista parametri)
{
        //corpul metodei
}
```

unde

acces=modificator de acces care stabileste care dintre celelalte parti ale programului pot apela metoda (este optonal). Daca lipseste, metoda este privata clasei in care a fost declarata. (Modificatorul de acces poate fi: public, protected, private, internal);

tip-rez= tipul rezultatului pe care metoda il intoarce (Ex: void, int, double…);

Nume=numele metodei, diferit de cuvintele cheie;

lista parametri =secventa de perechi separate prin virgula, in care primul element este un tip, iar al doilea un identificator.

Observatii: Exista doua moduri de a reveni dintr-o metoda

-intalnirea acoladei care marcheaza sfarsitul metodei (daca este de tip void);

-executia unei instructiuni return.

Daca metoda returneaza o valoare obtinuta in urma unui calcul sau ca reusita sau esec a unei operatii atunci se utilizeaza urmatoarea instructiune

```
return val;
```

Exemplul 2. In exemplul de mai jos este modificat programul din exemplul 1. Programul contine inca o clasa (clasa Line), iar codul care calcula distanta dintre doua puncte este organizat sub forma unei metode din clasa Line, numita Lung() .

```
using System;
class Point
{
    public double x;
    public double y;
}
class Line
{
    public Point punct1 = new Point();
    public Point punct2 = new Point();
    public double Lung()
    {
        double l;
        l= Math.Sqrt((punct1.x - punct2.x) *
      (punct1.x - punct2.x) + (punct1.y -
      punct2.y) * (punct1.y - punct2.y));
        return l;
    }
}
```

```
class Segmdr
{
    public static void Main()
    {
        Line seg = new Line();
        double dist;
        seg.punct1.x = 3;
        seg.punct1.y = 4;
        seg.punct2.x = 5;
        seg.punct2.y = 3;
        dist=seg.Lung();
          Console.WriteLine("Distanta dintre
      punctele ({0},{1}) si ({2},{3}) este:
      {4:#.##}", seg.punct1.x, seg.punct1.y,
      seg.punct2.x, seg.punct2.y, dist);
    }
}
```

Rezultat:

Distanta dintre punctele (3,4) si (5,3) este: 2,24

Metode.Utilizarea parametrilor

Parametru efectiv (argument)= valoare transmisa unei metode cand aceasta este invocata.

Parametru formal = variabila in corpul metodei care primeste valoarea parametrului efectiv.

Observatii:- parametrii formali se declara in interiorul parantezelor, dupa numele metodei;

-parametrii formali au domeniul de valabilitate corpul metodei;

-daca sunt folositi mai multi parametrii atunci fiecare isi specifica tipul. Acesta din urma poate fi diferit de tipul celorlalti.

Programul alaturat (Exemplul 3) testeaza daca primul intreg, notat prin d, este divizor al celui de-al doilea intreg. Ambii parametrii sunt introdusi de la tastatura.

Exemplul 3. a si b sunt paramerii formali, in timp ce d si m sunt parametri efectivi.

```
using System;
class Divizor
{
   public bool EsteDivizor(int a, int b)
   {
      if ((b % a) == 0)
         return true;
      else
         return false;
   }
}
class MyDiv
{
   public static void Main()
   {
      int d, m;
      Console.WriteLine("Programul testeaza daca d este divizor al lui m");
      Console.Write("Introduceti nr intreg d=");
      d = Convert.ToInt32(Console.ReadLine());
      Console.Write("\n Introduceti numarul intreg m=");
      m = int.Parse(Console.ReadLine());
      Divizor x=new Divizor();
      x.EsteDivizor(d, m);
      if (x.EsteDivizor(d, m))
        Console.WriteLine("{0} este divizor al lui {1}", d, m);
      else
        Console.WriteLine("{0} nu este divizor al lui {1}", d, m);
    }
}
```

Rezultat:

In functie de intregii introdusi de la tastatura, obtinem doua tipuri de mesaje.

Constructori

Variabilele instanta pot fi initializate manual (in Exemplul 1 variabilele x si y pentru ambele obiecte). Aceasta practica poate genera erori la initializarea unui camp. O solutie in acest sens o reprezinta utilizarea constructorilor.

Un constructor initializeaza un obiect atunci cand este creat. Au acelasi nume cu clasa din care fac parte. Nu au tip explicit.

Forma generala:

```
acces nume-clasa(parametri)
{
//codul constructorului
}
```

Se utilizeaza pentru atribuirea valorilor initiale pentru variabilele instanta definite in cadrul clasei sau pentru efectuarea altor operatii initiale necesare la crearea unui obiect complet initializat.

C# pune la dispozitie un constructor implicit care initializeaza toate variabilele membru cu zero (pt. tipuri valorice) respectiv cu null (pt. tipuri referinta). (Astfel, in Exemplul 1, imediat dupa crearea obiectelor punct1 si punct2, variabilele x si y ale ambelor obiecte au valorile 0. Apoi acestea sunt reinitializate manual). Daca este definit propriul constructor, cel implicit nu va fi utilizat.

Exemplul 4. Este modificat programul prezentat in exemplul 1 prin includerea unui constructor al clasei Point. La crearea obiectelor punct1 si punct2 se utilizeaza acest constructor care initializeaza variabilele x si y.

```
using System;
class Point
{
    public double x;
    public double y;
    public Point(double a, double b)
    {
        x = a;
        y = b;
     }
}
class Segmdr
{
    public static void Main()
    {
        Point punct1 = new Point(3,4);
        Point punct2 = new Point(5,3);
        double dist;
        dist = Math.Sqrt((punct1.x - punct2.x) * (punct1.x - punct2.x) + (punct1.y - punct2.y) * (punct1.y -
       punct2.y));
        Console.WriteLine("Distanta dintre punctele ({0},{1}) si ({2},{3}) este: {4:#.##}", punct1.x, punct1.y,
       punct2.x, punct2.y, dist);
    }
}
```

Rezultat:

Distanta dintre punctele (3,4) si (5,3) este: 2,24

Colectarea spatiului neutilizat

Utilizand operatorul new se aloca dinamic memorie libera pentru memorarea obiectelor create. Intrucat memoria nu este infinita, operatorul new poate sa esueze daca memoria este epuizata. Se pune astfel problema recuperarii memoriei din obiecte care nu mai sunt utilizate.

In cazul limbajului C++ se utilizeaza operatorul *delete* pentru a elibera memoria.

Limbajul C# pune la dispozitie o metoda diferita si anume colectarea automata a spatiului neutilizat.

Sistemul de colectare automata recupereaza spatiul ocupat de obiectele care nu mai sunt necesare programului. Daca nu mai exista nici o referinta la un obiect atunci se presupune ca acel obiect nu mai este necesar, iar memoria ocupata de el poate fi eliberata.

Colectarea automata se declanseaza destul de rar pe parcursul executiei programului. Nu se va declansa doar pentru ca exista obiecte care nu mai sunt folosite. Colectarea automata se declanseaza atunci cand se indeplinesc urmatoarele doua conditii: exista o mare necesitate de memorie si exista obiecte care pot fi reciclate.

Colectarea necesita un timp mare. Astfel aceasta se va declansa doar daca este imperios necesar. Nu se poate determina cu exactitate cand are loc colectarea spatiului neutilizat

Destructori

Destructorii au forma generala:

~nume-clasa()
{
// codul destructorului
}

Destructorul este deci declarat similar cu un constructor, fiind insa precedat de caracterul ~(tilda).

Pentru adaugarea unui destructor la o clasa, il includeti ca si pe orice alt membru al clasei. Acesta va fi apelat inainte ca un obiect din acea clasa sa fie reciclat. In interiorul destructorului, se vor specifica actiunile care trebuie executate inainte ca obiectul sa fie distrus.

Destructorul se apeleaza imediat inaintea colectarii automate. Acesta nu va fi apelat cand se iese din domeniul de valabilitate al obiectului. Lucrul acesta difera de C++, unde destructorii sunt apelati cand domeniul de valabilitate se incheie.Nu se poate determina cu exactitate cand se va executa un destructor.

Mai mult, este posibil ca programul dumneavoastra sa se termine inaintea inceperii operatiei de colectare automata, caz in care destructorul nu va fi apelat deloc.

Exemplul 5. Utilizarea destructorilor

```
using System;
class Destructor
{
    int x;
    public Destructor(int i)
    {
        x=i;
    }
    ~Destructor()
    {
        Console.WriteLine("Distrugem "+x);
    }

    public void generator(int i)
    {
        Destructor o = new Destructor(i);
    }

}
class DestrDemo
{
    public static void Main()
    {
        int num;
        Destructor ob=new Destructor(0);
        for (num = 1; num < 10000; num++)
            ob.generator(num);
    }
}
```

Tablouri

Definitie. Un tablou reprezinta o colectie de variabile de acelasi tip, referite prin intermediul unui nume comun.

Observatii:

- La fel ca si in alte limbaje de programare, in C# tablourile pot avea mai multe dimensiuni;
- Tablourile se pot utiliza ca si tablourile din alte limbaje de programare, insa spre deosebire de acestea, in C# tablourile sunt implementate ca obiecte.
- Un avantaj obtinut prin implementarea tablourilor ca obiecte este acela ca spatiul ocupat de obiectele neutilizate poate fi colectat automat.

Declararea unui tablou unidimensional:

tip [ ] nume_tablou = new tip[dimensiune];

unde tip reprezinta tipul de baza al tabloului, nume_tablou reprezinta un identificator care contine referinta la zona de memorie alocata de new, iar dimensiune reprezinta numarul de elemente pe care tabloul le memoreaza. (Ex: int [ ] n=new int[6]; prima parte a instructiunii declara variabila de referinta n la tablou. In partea a doua se utilizeaza operatorul new pentru a aloca dinamic (la executia programului) memorie pentru tablou si pentru a atribui variabilei n o referinta la zona de memorie alocata).

Tabloul fiind un obiect, putem separa declaratia de mai sus in doua parti:

tip [ ] nume_tablou;

nume_tablou= new tip[dimensiune];

Accesarea unui element din tablou:

Un element individual din tablou poate fi accesat utilizand un index. Indexul descrie pozitia unui element din tablou. La fel ca in limbajul C, in C# toate tablourile au indexul primului element egal cu zero. In cazul exemplului de mai sus, primul element este n[0], al doilea este n[1],… cel de-al saselea este n[5].

Exemplul 1. In programul de mai jos sunt determinate cel mai mic element si cel mai mare element ale unui tablou ale carui elemente de tip int sunt initializate manual.

```
using System;
class MinMax
{
    public static void Main()
    {
        int[] n = new int[6];
        int min, max;
        n[0] = 3298;
        n[1] = 8;
        n[2] = -98;
        n[3] = 48;
        n[4] = -298;
        n[5] = -28;
        min = max = n[0];
        for (int i = 1; i < 6; i++)
        {
            if (n[i] < min)
            {
                min = n[i];
            }
            if (n[i] > max)
            {
                max = n[i];
            }
        }
        Console.WriteLine("min={0}, max={1}", min, max);
    }
}
```

Rezultat:

min=-298, max=3298

Initializarea unui tablou:

In programul anterior, valorile elementelor individuale au fost incarcate manual. O modalitate mai simpla este aceea de a realiza aceasta operatie direct de la creare.

Forma generala pentru initializarea unui tablou este:

```
tip [  ] nume_tablou={val1, val2, val3,… valN};
```

unde elementelor nume_tablou[0], nume_tablou[1],… nume_tablou[N-1] ale tabloului le-au fost atribuite valorile val1, val2,…si respectiv valN.

Observatii:

- Cand un tablou se initializeaza in acest mod nu este necesara utilizarea operatorului new;
- Desi redundanta, este corecta utilizarea formei

```
tip [  ] nume_tablou=new tip[ ] {val1, val2, val3,… valN};
```

- De asemenea, putem separa instructiunea de mai sus in doua comenzi:

```
tip [  ] nume_tablou;
nume_tablou=new tip[ ] {val1, val2, val3,… valN};
```

- Este GRESITA insa utilizarea instructiunilor:

```
tip [  ] nume_tablou;
nume_tablou= {val1, val2, val3,… valN};
```

- Este corecta si utilizarea formei

```
tip [  ] nume_tablou=new tip[N] {val1, val2, val3,… valN};
```

insa dimensiunea N a tabloului trebuie sa fie egala cu numarul elementelor initializate. Mai mult N trebuie sa fie un literal. Astfel, este corecta initializarea

```
const int arraySize=5;
int [ ] myArray=new int [arraySize] {0, 2, 4, 6, 8};
```

insa urmatoarele sunt gresite:

```
int arraySize=5;
int [ ] myArray=new int [arraySize] {0, 2, 4, 6, 8};
```

```
const int arraySize=5;
int [ ] myArray=new int [arraySize] {0, 2, 4};
```

Verificarea marginilor

Marginile tablourilor sunt strict verificate. Depasirea sfarsitului de tablou sau utilizarea indicilor negativi genereaza erori la executie. Spre exemplu: programul de mai jos, dupa ce variabila i atinge valoarea 50 genereaza o exceptie de tipul IndexOutOfRangeException si programul se termina:

Exemplul 2. Programul genereaza o eroare la executie.

```
using System;
class ArrayErr
{
    public static void Main()
    {
  int[ ] array = new int[50];
        for (int i = 0; i < 100; i++)
          {
            array[i] = i;
             Console.WriteLine("array[{0}]={1}", i,array[i]);
          }
    }
}
```

Tablouri multidimensionale

Forma generala a unui tablou multidimensional este:

```
tip [ ,…, ] nume_tablou = new tip[dim1, dim2, … dim N];
```

Spre exemplu, declaratia de mai jos creaza un tablou tridimensional de elemente de tip long cu dimensiunile 6x7x12

```
long [ , , ] n=new long[6,7,12];
```

La fel ca in cazul unui tablou unidimensional, initializarea tabloului se poate face fie manual (spre exemplu instructiunea n[3,5,10]=76; atribuie elementului avand indexul [3,5,10] valoarea 76), fie la crearea tabloului. In acest caz, initializarea se face prin includerea listei de initializare a fiecarei dimensiuni intr-o pereche proprie de acolade. Daca avem de-a face cu un tablou bidimensional, pentru care primul index variaza de la 0 la M-1, iar cel de-al doilea de la 0 la N-1, atunci initializarea se face astfel

tip [,] nume_tablou={ {$val_0$0, val_0 1,… val_0N-1}, {val_1 0, val_1 1,… val_1 N-1}, …
{val_{M-1} 0, val_{M-1} 1,… val_{M-1} N-1}};

Observatie: Blocurile sunt separate prin virgule, iar acolada finala este urmata de punct si virgula.

Exemplele 3 si 4

```
/*initializarea si afisarea valorilor
 unui tablou bidimensional pentru care
 prima dimensiune este 2 iar cea de-a doua
 dimensiune este 3 */

using System;
class ArrayErr
{
    public static void Main()
    {
        int[ , ] array = new int[ , ] {{1,2,3},
                                       {3,5,6}};
        for (int i = 0; i < 2; i++)
            for(int j=0; j<3; j++)
        {
            Console.WriteLine("array[{0},{1}]={2}", i,j,
       array[i,j]);
        }
    }
}
```

Rezultat:
array[0,0]=1
array[0,1]=2
array[0,2]=3
array[1,0]=3
array[1,1]=5
array[1,2]=6

```
/*initializarea si afisarea valorilor
 unui tablou tridimensional avand
 dimensiunile 3, 4 si respectiv 2 */

using System;
class ArrayErr
{
    public static void Main()
    {
  int[, ,] array ={ {{2,3}, {5,6}, {1,-1}, {0,6}},
              {{4,5}, {6,-3}, {3,-5}, {8,9}},
     {{-2,33}, {15,16},{11,-10}, {10,60}} };
      for (int i = 0; i < 3; i++)
          for(int j=0; j<4; j++)
              for (int k=0; k<2; k++)
              {
              Console.WriteLine("array[{0},{1},{2}]={3}",
      i,j,k, array[i,j,k]);
              }
    }
}
```

Tablouri in scara

In exemplele anterioare in care au fost create tablouri bidimensionale, au fost create asa numitele *tablouri dreptunghiulare*. Adica toate liniile au avut acelasi numar de elemente.

In cazul in care doriti sa creati un tablou bidimensional in care lungimea fiecarei linii sa fie diferita, puteti utiliza asa numitele *tablouri in scara.*

Definitie. Tablourile in scara sunt tablouri cu elementele tablouri.

Forma generala utilizata pentru declararea unui tablou in scara este:

tip[ ] [ ] nume_tablou=new tip [dim] [ ];

unde dim reprezinta numarul de linii din tablou. Liniile nu au fost inca alocate. Acestea se aloca in mod individual, pentru ca lungimea liniei sa varieze. Forma generala este:

nume_tablou[0]=new tip[dim_0];

nume_tablou[1]=new tip[dim_1];

…………………………………….

nume_tablou[dim-1]=new tip[dim_{dim-1}];

Exemplu: int [ ] [ ] array=new int [2] [ ]; //aceasta secventa de cod aloca memorie pentru

```
array[0]=new int[3];            //un tablou avand doua linii. Prima linie contine 3
array[1]=new int[7];            //elemente, iar cea de-a doua 7 elemente.
```

Majoritatea aplicatiilor nu utilizeaza tablourile in scara. Insa aceste sunt utile in unele situatii, spre exemplu atunci cand este necesara memorarea elementelor unei matrici de dimensiuni foarte mari, insa care are doar putine elemente semnificative.

Exemplul 5. Utilizarea tablourilor in scara

```
using System;
class ArrayErr
{
    public static void Main()
    {
        int[][] pasageri =new int[7][];
        pasageri[0]=new int[4]{20,14,8,9};
        pasageri[1] = new int[4] { 14, 7, 14, 8 };
        pasageri[2] = new int[4] { 17, 12, 9, 19 };
        pasageri[3] = new int[4] { 13, 15, 18, 14 };
        pasageri[4] = new int[4] { 20, 19, 18, 20 };
        pasageri[5] = new int[2] { 10, 5 };
        pasageri[6] = new int[2] { 7, 9 };
           for (int i = 0; i < 5; i++)
            for(int j=0; j<4; j++)
                {
                Console.WriteLine("In ziua {0}, cursa {1} a avut {2} pasageri", i+1,j+1, pasageri[i][j]);
                }
           for (int i = 5; i < 7; i++)
               for (int j = 0; j < 2; j++)
               {
                   Console.WriteLine("In ziua {0}, cursa {1} a avut {2} pasageri", i + 1, j + 1, pasageri[i][j]);
               }
    }
}
```

Rezultat: Programul afiseaza numarul de pasageri pe care l-a avut fiecare cursa.

Fiecarui tablou ii este asociata proprietatea Length.

Aceasta contine numarul de elemente pe care tabloul le poate memora.

Exemplul 6. Utilizarea proprietatii Length

```
using System;
class LenghtDemo
{
 public static void Main()
{
 int[] list = { 2, 4, 6, 4, 3, 8, 6, 9 };
 int[][] pasageri =new int[7][];
 pasageri[0]=new int[4]{20,14,8,9};
 pasageri[1] = new int[4] { 14, 7, 14, 8 };
 pasageri[2] = new int[4] { 17, 12, 9, 19 };
 pasageri[3] = new int[4] { 13, 15, 18, 14 };
 pasageri[4] = new int[4] { 20, 19, 18, 20 };
 pasageri[5] = new int[2] { 10, 5 };
 pasageri[6] = new int[2] { 7, 9 };
 Console.WriteLine("Lungimea tabloului list este {0}", list.Length );
 Console.WriteLine("Lungimea tabloului pasageri este {0}", pasageri.Length);
 Console.WriteLine("Lungimea tabloului pasageri[0] este {0}", pasageri[0].Length);
 Console.WriteLine("Lungimea tabloului pasageri[5] este {0}", pasageri[5].Length);
    Console.Write("Tabloul list:\t");
 for (int i = 0; i < list.Length ; i++)
    Console.Write(list[i] + " ");
 Console.WriteLine();
 }
}
```

Rezultat:

Lungimea tabloului list este 8
Lungimea tabloului pasageri este 7
Lungimea tabloului pasageri[0] este 4
Lungimea tabloului pasageri[5] este 2
Tabloul list:2 4 6 4 3 8 6 9

Bucla *foreach*

Bucla *foreach* se utilizeaza pentru ciclarea prin elementele unei colectii. O colectie reprezinta un grup de obiecte. C# defineste mai multe tipuri de colectii, unul dintre acestea o reprezinta tablourile.

Forma generala a buclei foreach este:

```
foreach(tip nume_var in colectie)
{instructiuni; }
```

unde tip nume_var specifica tipul si numele unei variabile de iterare care va primi la fiecare iteratie a lui foreach valoarea unui element din colectie. Colectia prin care se face ciclarea este specificata de colectie.

Observatie: Variabila de iterare trebuie sa fie de acelasi tip sau de un tip compatibil cu tipul de baza al colectiei.

Exemplul 7. Utilizarea buclei foreach pentru ciclarea intr-un tablou unidimensional.

```
using System
class ForeachDemo
{
   public static void Main()
   {
      int sum=0;
      int [] n =new int[20];

      for (int i = 0; i < n.Length; i++)
         n[i] = i*i;

      foreach (int x in n)
      {
         Console.WriteLine("valoarea este: "
    + x);
         sum +=x;
      }

      Console.WriteLine("Suma este {0}",
    sum );

   }
}
```

Rezultat:

```
valoarea este: 0
valoarea este: 1
valoarea este: 4
valoarea este: 9
valoarea este: 16
valoarea este: 25
valoarea este: 36
valoarea este: 49
valoarea este: 64
valoarea este: 81
valoarea este: 100
valoarea este: 121
valoarea este: 144
valoarea este: 169
valoarea este: 196
valoarea este: 225
valoarea este: 256
valoarea este: 289
valoarea este: 324
valoarea este: 361
Suma este 2470
```

Observatie: Principala diferenta intre instructiunile for si foreach este accea ca foreach ofera acces doar la citirea elementelor unei colectii. Spre exemplu, instructiunea:

```
int [ ] n = new int[3];
for (int i = 0; i < n.Length; i++)
    n[i] = 5*5;
```

nu poate fi inlocutita cu:

```
foreach (int x in n)
    x = 5*5;
```

sau orice alta comanda care incearca sa atribuie valori elementelor colectiei.

Stringuri

Stringuri

Unul dintre cele mai importante tipuri de date este tipul string. Tipul string defineste si implementeaza sirurile de caractere. Spre deosebire de alte limbaje de programare unde stringurile sunt reprezentate ca tablouri de caractere, in C# stringurile sunt obiecte. Asadar, tipul string este un *tip de referinta*.

Observatii: -un string poate fi construit prin utilizarea unui literal, spre exemplu:

string str="acesta este un sting";

sau prin pornind de la un tablou de tip char, spre exemplu:

char [ ] charray={'a', 'b', 'w', 'r'};

string str=new string(charray);

- clasa string contine mai multe metode care opereaza asupra stringurilor. Spre exemplu:

static string Copy(string str); intoarce o copie a stringului str;

int CompareTo(string str) intoarce o valoare negativa daca stringul care invoca metoda este mai mic decat str, pozitiva daca este mai mare si nula daca stringurile sunt egale;

int indexOf(string str) cauta in stringul apelant subsirul str. Intoarce indicele primei aparitii, sau -1 in caz de esec;

int LastindexOf(string str) cauta in stringul apelant subsirul str. Intoarce indicele ultimei aparitii, sau -1 in caz de esec;

string Substring(int startindex, int len) intoarce un substring al stringului apelant. Parametrul startindex precizeaza indicele de inceput, iar len lungimea subsirului extras;

string [ ] Split(char [ ] separator, int count) intoarce un tablou de tip string avand dimensiunea mai mica sau egala cu count. Tabloul este obtinut prin impartierea stringului apelant in subsiruri. Aceasta operatie se realizeaza la intalnirea separatorului specificat de separator.

-tipul string contine proprietatea Length;

-pentru determinarea valorii unui caracter dintr-un string se utilizeaza un index. Exemplu:

```
string str="sirdecaractere";
Console.WriteLine(str[3]);
```

Metoda WriteLine va afisa litera d;

-pentru a verifica daca doua stringuri sunt egale se poate utiliza operatorul ==. In cazul in care operatorul == este aplicat asupra referintelor la obiecte, acesta stabileste daca cele doua referinte refera acelasi obiect. In cazul stringurilor, chiar daca acestea sunt obiecte, operatorul == compara efectiv daca cele doua stringuri sunt egale. Acelasi lucru se intampla si cu operatorul != care compara continutul a doua obiecte de tip string.

-pentru a concatena doua stringuri se utilizeaza operatorul +;

-stringurile nu se pot modifica. Dupa creare, un obiect de tip string nu poate fi modificat. Daca este nevoie de un string care este o variatie a unui string deja existent atunci trebuie creat unul nou care sa contina modificarile dorite.

Exemplul 1. Metode si operatii cu stringuri

```
using System;
class StringDemo
{  public static void Main()
   {   string str1="Acesta este un string";
       string str2=string.Copy(str1);
       string str3 = "Acesta este un alt string";
       for (int i = 0; i < str1.Length; i++)
          Console.Write(str1[i]);
       Console.WriteLine();
       if (str1 == str2)
          Console.WriteLine("str1=str2");
       else
          Console.WriteLine("str1 !=str2");
       if (str1 == str3)
          Console.WriteLine("str1=str3");
       else
          Console.WriteLine("str1 !=str3");
       int result = str1.CompareTo(str3);
          if(result ==0)
             Console.WriteLine("str1 este egal cu
      str3");
          else
             Console.WriteLine("str1 si str3 sunt
      diferite");
       string s = str3.Substring(12, 13);
       Console.WriteLine(s);
   }
}
```

Rezultat:

Acesta este un string
str1=str2
str1 !=str3
str1 si str3 sunt diferite
un alt string

```
using System;
class MinMax
{
   public static void Main()
   {
      string S = "Acesta este un string";
      string subsirS;
      subsirS = S.Substring(0, 8);
      Console.WriteLine(subsirS);
   }
}
```

Rezultat:

Acesta e

```
using System;
class MinMax
{
   public static void Main()
   {
      string S = "Acesta este un string";
      string subsirS = "est";
      int i,j;
      i = S.IndexOf(subsirS);
      j = S.LastIndexOf(subsirS);
      Console.WriteLine("Primul index este {0}",i);
      Console.WriteLine("Ultimul index este {0}", j);
   }
}
```

Rezultat:

Primul index este 2
Ultimul index este 7

Exemplu:

```
using System;
class Citire
{
    public static void Main()
    {
        string [ ] myStringArray;
        char[ ] charArray = { '-' };

        string myString="This-is-my-string";
        myStringArray=myString.Split(charArray,2);
        foreach (string s in myStringArray)
        {
            Console.WriteLine(s);
        }
    }
}
```

Rezultat:

This
is-my-string

O privire detaliata asupra claselor si metodelor

Specificatorii de acces din C#

Controlul accesului la membrii unei clase se realizeaza prin utilizarea urmatorilor specificatori de acces:

- *public* (membrii publici pot fi accesati liber de codul din afara clasei);

-*private* (membrii privati sunt accesibili numai metodelor definite in aceeasi clasa);

-*protected* (membrii protejati pot fi accesati de metodele definite in cadrul aceleiasi clase sau de metodele definite in cadrul claselor care mostenesc clasa data);

-*internal* (specificatorul internal este utilizat pentru a declara membrii care sunt cunoscuti in toate fisierele dintr-un asamblaj, insa nu in afara asamblajului).

-*protected internal* (membrii sunt vizibili atat in clasele care mostenesc clasa in care sunt definiti acesti membrii cat si in cadrul grupului de fisiere care formeaza asamblajul).

Observatie: nu exista specificatori de tipul public internal sau private internal.

Exemplul 2. Utilizarea specificatorilor de acces

```
using System;
class Persoana
{
    protected string nume;
    protected string prenume;
     public Persoana(string nm, string pnm)
     {
     nume = nm;
     prenume = pnm;
     }
 }
     class Angajat : Persoana
     {
     private int anulAngajarii;
     public Angajat(string nm, string pnm, int anang )
       : base(nm, pnm)
     {
     anulAngajarii = anang;
     }
     public void
       AfiseazaNumeleIntregSiAnulAngajarii()
     {
     Console.WriteLine("Angajat: {0} {1} {2}", nume,
       prenume, anulAngajarii);
     }
     }
```

```
class NameApp
     {
     public static void Main()
     {
     Angajat el = new Angajat("Popescu", "Ion",
       1983);
     el.AfiseazaNumeleIntregSiAnulAngajarii();
     }
     }
```

Rezultat:

Angajat: Popescu Ion 1983

Modul de transfer al parametrilor catre metode

Ca parametri pentru metode pot fi utilizati tipurile valorice (int, double, etc.), insa pot fi transmise si obiecte.

Exemplul 3. Transmiterea obiectelor ca parametri pentru metode

```
using System;
class Dreptunghi
{
   int l, L;
   int arie;
      public Dreptunghi(int i, int j)
   {
      l = i;
      L = j;
      arie = l * L;
   }
   public bool Congruent(Dreptunghi obiect)
   {
      if ((obiect.l == l) & (obiect.L == L))
         return true;
      else
         return false;
   }
   public bool Echivalent(Dreptunghi obiect)
   {
      if (obiect.arie == arie)
         return true;
      else
         return false;
   }
```

```
class Transmob
   {
      public static void Main()
      {
         Dreptunghi obiect1 = new Dreptunghi(2, 3);
         Dreptunghi obiect2 = new Dreptunghi(2, 3);
         Dreptunghi obiect3 = new Dreptunghi(1, 6);
         Console.WriteLine("Afirmatia: <<obiectul1 este congruent cu obiectul2>> este {0}", obiect1.Congruent(obiect2));
         Console.WriteLine("Afirmatia << obiectul1 este congruent cu obiectul3>> este  {0}", obiect1.Congruent(obiect3));
         Console.WriteLine("Afirmatia <<obiectul1 este echivalent cu obiectul3>> este {0}", obiect1.Echivalent(obiect3));
      }
   }
}
```

Rezultat:

Afirmatia: <<obiectul1 este congruent cu obiectul2>> este True

Afirmatia << obiectul1 este congruent cu obiectul3>> este False

"Afirmatia <<obiectul1 este echivalent cu obiectul3>> este True

In C# exista doua modalitati de transmitere al parametrilor catre metode si anume: *transferul prin valoare* si *transferul prin referinta*. In mod implicit, tipurile valorice sunt transferate metodelor prin valoare, in timp ce obiectele sunt transferate prin referinta.

Pentru a vedea care este diferenta dintre aceste doua modalitati sa consideram mai intai urmatoarele secvente de cod:

a) int a; int b; a=10; b=a;

b) Dreptunghi ob1; Dreptunghi ob2; ob1=new Dreptunghi(2,7); ob2=ob1;

In cazul a) se declara mai intai doua variabile a si b. Prima dintre ele se initializeaza cu valoarea 10 dupa care si prima variabila primeste aceeasi valoare. Asadar au fost create doua variabile, ambele au valoarea 10.

In cazul b) se declara doua variabile ob1 si ob2. Apoi, utilizand operatorul new, se creaza o imagine fizica a unui obiect care este referit prin intermediul variabilei ob1. Ultima instructiune face ca si ob2 sa refere acelasi obiect. Asadar a fost creat un singur obiect referit prin intermediul a doua variabile ob1 si ob2.

Aceasta diferenta face ca cele doua modalitati de transfer (prin valoare si respectiv prin referinta) ale parametrilor sa aiba nuante diferite. Astfel:

-*transferul prin valoare*: Metoda copiaza valoarea parametrului efectiv in parametrul formal al subrutinei. Modificarile aduse parametrului subrutinei nu vor modifica valoarea parametrului efectiv.

-*transferul prin referinta*: Se transmite parametrului formal o referinta a parametrului efectiv si nu valoarea acestuia. In interiorul subrutinei, referinta este utilizata pentru accesul la parametrul efectiv. Asadar modificarile parametrului formal vor afecta si parametrul efectiv.

Exemplul 4. Transferul prin valoare si transferul prin referinta

```
using System;
class Test
{
    public int a, b;

    public Test(int i, int j)
    {
        a=i;
        b=j;
    }
}
class DemoReferintaValoare
{
    public static void NoChange(int i, int j)
    {
        i = i + j;        j = -j;
    }
    public static void Change(Test obiect)
    {
        obiect.a = obiect.a + obiect.b;
        obiect.b = -obiect.b;
    }
```

```
public static void Main()
    {
        Test ob = new Test(10,20);
        Console.WriteLine("a={0} si b={1}
      inainte de apelul metodei
      NoChange",ob.a,ob.b);

        NoChange(ob.a,ob.b);

        Console.WriteLine("a={0} si b={1}
      dupa apelul metodei NoChange", ob.a,
      ob.b);
        Console.WriteLine("a={0} si b={1}
      inainte de apelul metodei Change",
      ob.a, ob.b);

        Change(ob);

        Console.WriteLine("a={0} si b={1}
      dupa apelul metodei Change", ob.a,
      ob.b);
    }
}
```

Rezultat:

a=10 si b=20 inainte de apelul metodei NoChange
a=10 si b=20 dupa apelul metodei NoChange
a=10 si b=20 inainte de apelul metodei Change
a=30 si b=-20 dupa apelul metodei Change

Utilizarea modificatorilor ref si out

In mod implicit tipurile valorice sunt transferate catre metode prin valoare. Insa acest comportament poate fi modificat cu ajutorul cuvintelor cheie ref si out.

Modificatorul de parametrii ref:

-se utilizeaza atunci cand se doreste ca o anumita metoda sa modifice parametrii efectivi;

-forteaza in C# transferul prin referinta in detrimentul transferului prin valoare;

-apare atat in declaratia cat si in apelul metodei. Parametrul transferat cu ref trebuie sa aiba o valoare atribuita inainte de apel. Asadar, nu se poate apela o metoda care initializeaza un parametru avand modificatorul ref.

Exemplul 5. Utilizarea parametrului ref

```
using System;
class DemoShimb
{
    public static void Schimb(ref int i, ref int j)
    {
        int t;
        t = i;
        i = j;
        j = t;
    }
    public static void Main()
    {
        int x=10, y=20;
        Console.WriteLine("x={0} si y={1} inainte de apelul metodei Schimb",x,y);
        Schimb(ref x,ref y);
        Console.WriteLine("x={0} si y={1} dupa apelul metodei Schimb", x, y);
    }
}
```

Rezultat:

x=10 si y=20 inainte de apelul metodei Schimb

x=20 si y=10 dupa apelul metodei Schimb

Modificatorul de parametrii out:

-se utilizeaza atunci cand se doreste intoarcerea de catre metoda a mai multor valori mediului apelant. O instructiune return intoarece o singura valoare. Daca se doreste intoarcerea spre exemplu a doua valori atunci problema se rezolva cu ajutorul modificatorului out.

-apare atat in declaratia cat si in apelul metodei;

-intoarce o valoare dintr-o metoda. Nu este necesar ca variabila utilizata ca parametru out sa fie initializata inainte de apelul metodei. Metoda va da acesteia o valoare. Mai mult, in corpul metodei, un parametru out este considerat ca neinitializat.

Exemplul 6. Utilizarea parametrului out

```
using System;
class Dreptunghi
{
    int l;
    int L;
    public Dreptunghi(int i, int j)
    {
        l = i;
        L = j;
    }
    public int InfoDreptunghi_si_Arie(out
     bool patrat)
    {
        if (l == L)
            patrat = true;
        else
            patrat = false;
        return l * L;
    }
}
```

```
class DemoDreptunghi
{
    public static void Main()
    {
        int aria;
        bool p;
        Dreptunghi ob = new Dreptunghi(5, 10);
        aria = ob.InfoDreptunghi_si_Arie(out p);
        Console.WriteLine("Dreptunghiul este
     patrat: {0}",p);
        Console.WriteLine("Aria figurii
     geometrice considerate este {0}",aria);
    }
}
```

Rezultat:

Dreptunghiul este patrat: False

Aria figurii geometrice considerate este 50

Utilizarea unui numar variabil de parametri

La crearea unei metode, numarul parametrilor transmisi metodei este in general cunoscut. Exista situatii in care acest numar este necunoscut.

In acest caz se utilizeaza modificatorul params pentru a declara un tablou de parametri. Acest tablou poate contine 0, 1 sau mai multi parametri.

In cazul in care o metoda are atat parametri obisnuiti cat si un parametru params acesta din urma trebuie sa fie ultimul din lista de parametri.

In plus, nu poate exista mai mult de un parametru de tipul params.

Exemplul 7.Programul de mai jos calculeaza maximul dintr-o secventa de valori

```
using System;
class Maxim
{
    public int ValMax(params int [  ] nume)
    {
        int m;
        if (nume.Length == 0)
        {
            Console.WriteLine("Eroare: nu
      sunt parametri");
            return 0;
        }
else
        {
            m = nume[0];
            for (int i = 0; i < nume.Length; i++)
            {
                if (nume[i] > m)
                    m = nume[i];
            }
            return m;
        }
    }
}
```

```
class DemoMaxim
{
    public static void Main()
    {
        int max;
        Maxim ob = new Maxim();
        int[ ] tablou = { 0, 3, 5, 7, -23, 44 };
        max=ob.ValMax(tablou);
        Console.WriteLine("Maximul este {0}",max);

        int a = 7, b = 22;
        max = ob.ValMax(a, b);
        Console.WriteLine("Maximul dintre {0} si {1}
      este: {2}", a,b, max);

    }
}
```

Rezultat:

Maximul este 44

Maximul dintre 7 si 22 este: 22

Supraincarcarea metodelor

Una dintre cele mai interesante facilitati oferite de C# o reprezinta supraincarcarea metodelor.

Definitie. Spunem ca metodele sunt supraincarcate atunci cand doua sau mai multe metode din cadrul aceleiasi clase au acelasi nume.

Observatii: -Daca metodele sunt supraincarcate atunci declaratiile parametrilor lor difera. Metodele au un numar diferit de parametri sau acelasi numar de parametrii insa de tipuri diferite. Pe scurt, metodele nu au aceeasi *signatura. Signatura* unei metode reprezinta numele unei metode si lista parametrilor acesteia.

-Signatura unei metode nu include tipul rezultatului intors. Daca metodele au acelasi nume, acelasi numar si tip de parametrii insa tipul rezultatului intors este diferit atunci se produce o eroare. Compilatorul nu dispune de suficienta informatie pentru a decide care metoda sa aleaga.

-Prin supraincarcarea metodelor, limbajul C# implemeteaza conceptul de *polimorfism*. Altfel spus, C# implementeaza *paradigma "o singura interfata, mai multe metode*". In limbajele care nu permit supraincarcarea, exista mai multe versiuni ale aceleiasi functii. Spre exemplu, in C, functia abs( ) intoarce valoarea absoluta a unui intreg, labs( ) valoarea absoluta a unui intreg lung, iar fabs( ) valoarea absoluta a unui float. In C# toate metodele care calculeaza valoarea absoluta au acelai nume Abs( ). Este misiunea compilatorului sa aleaga care metoda a lui Abs( ) trebuie sa o foloseasca. Asadar, utilizand conceptul de supraincarcare a metodelor, mai multe metode au fost comprimate in una singura.

-Exista situatii in care tipurile parametrilor formali nu coincid cu tipurile parametrilor efectivi. In acest caz se realizeaza o conversie automata de tip. Versiunea metodei alese este aceea pentru care setul de parametrii formali este cel mai apropiat de setul de parametri efectivi.

Exemplul 8: Supraincarcarea metodelor

```
using System;
class Overload
{
   public void Ovload()
   {
      Console.WriteLine("Nici un parametru");
   }
   public void Ovload(int a)
   {
      Console.WriteLine("Un parametru de tip int:
    a={0}",a);
   }
   public void Ovload(double a)
   {
      Console.WriteLine("Un parametru de tip
    double: a={0}", a);
   }
   public void Ovload(ref double a)
   {
      Console.WriteLine("Un parametru de tip ref
    double: a={0}", a);
   }
   public int Ovload(int a, int b)
   {
      Console.WriteLine("Doi parametrii de tip int:
    a={0} b={1}", a, b);
      return a + b;
   }
public double Ovload(double a, double b, double c)
   {
      Console.WriteLine("Trei parametrii de tip double:
    a={0}, b={1}, c={2}", a,b,c);
      return a*b*c;
   }
}

class DemoOverload
{
   public static void Main()
   {
      double x = 3.1;
      Overload ob = new Overload();
      ob.Ovload();
      ob.Ovload(2);
      ob.Ovload(2.1);
      ob.Ovload(ref x);
      ob.Ovload(2, 3);
      ob.Ovload(2, 3, 4);
   }
}
```

Rezultat:

Nici un parametru
Un parametru de tip int: a=2
Un parametru de tip double: a=2,1
Un parametru de tip ref double: a=3,1
Doi parametrii de tip int: a=2 b=3
Trei parametrii de tip double: a=2, b=3, c=4

Exemplul 9: Supraincarcarea metodelor

```
using System;
class Overload{
   public double Aria(double r)    {
      double A=4*(Math.Atan(1))*r*r;
      Console.WriteLine("Aria cercului avand raza r={0} este A={1:#.##}", r, A);
      return A;
   }
   public double Aria(double b, double h)  {
      double A;
      A = h * b/2;
      Console.WriteLine("Aria triunghiului avand baza b={0} si inaltimea h={1} este A={2}",b,h,A);
      return A;
   }
   public double Aria(double b, double B, double h)   {
      double A;
      A = (b+B) * h / 2;
      Console.WriteLine("Aria trapezului avand baza mica b={0}, baza mare B={1} si inaltimea h={2} este A={3}",
     b, B, h, A);
      return A;
   }
}
class DemoOverload{
     public static void Main()
   {
      Overload ob = new Overload();
      ob.Aria(3);        ob.Aria(2,3);       ob.Aria(2,3,4);
   }
}
```

Rezultat:

Aria cercului avand raza r=3 este A=28,27

Aria triunghiului avand baza b=2 si inaltimea h=3 este A=3

Aria trapezului avand baza mica b=2, baza mare B=3 si inaltimea h=4 este A=10

Exemplul 10. Supraincarcarea constructorilor

```
using System;                          //Ca orice alte metode, constructorii pot fi si ei supraincarcati
class Point {
    public double x;
    public double y;
    public Point()    {    }
    public Point(double a)    {   x = a;  }
    public Point(double a, double b)    {   x = a;    y = b;   }
}
class Segmdr{
    public static void Main()    {
        Point punct1 = new Point();
        punct1.x = 3;     punct1.y = 5;
        Point punct2 = new Point(10);
        punct2.y = 5;
        Point punct3 = new Point(2,3);
        double dist;
        dist = Math.Sqrt((punct1.x - punct2.x) * (punct1.x - punct2.x) + (punct1.y - punct2.y) * (punct1.y - punct2.y));
        Console.WriteLine("Distanta dintre punctele ({0},{1}) si ({2},{3}) este: {4:#.##}", punct1.x, punct1.y, punct2.x,
       punct2.y, dist);
        dist = Math.Sqrt((punct1.x - punct3.x) * (punct1.x - punct3.x) + (punct1.y - punct3.y) * (punct1.y - punct3.y));
        Console.WriteLine("Distanta dintre punctele ({0},{1}) si ({2},{3}) este: {4:#.##}", punct1.x, punct1.y, punct3.x,
       punct3.y, dist);
        dist = Math.Sqrt((punct2.x - punct3.x) * (punct2.x - punct3.x) + (punct2.y - punct3.y) * (punct2.y - punct3.y));
        Console.WriteLine("Distanta dintre punctele ({0},{1}) si ({2},{3}) este: {4:#.##}", punct2.x, punct2.y, punct3.x,
       punct3.y, dist);
    }
}
```

Rezultat:

Distanta dintre punctele (3,5) si (10,5) este: 7
Distanta dintre punctele (3,5) si (2,3) este: 2,24
Distanta dintre punctele (10,5) si (2,3) este: 8,25

Metoda Main

Pana acum a fost utilizata o singura forma a metodei Main( ). Exista insa mai multe forme ale acesteia datorate supraincarcarii. Astfel:

-Metoda Main( ) poate intoarce o valoare in procesul apelant (in sistemul de operare). In acest caz se utilizeaza urmatoarea forma a metodei Main( ):

```
public static int Main( );
```

De regula, valoarea intoarsa de Main( ) indica daca programul s-a terminat normal sau ca urmare a unei conditii de eroare. Prin conventie, daca valoarea intoarsa este 0 atunci programul s-a terminat normal. Orice alta valoare indica faptul ca s-a produs o eroare.

-Metoda Main( ) poate accepta parametrii. Acesti parametrii se mai numesc parametrii in linia de comanda. Un parametru in linia de comanda reprezinta o informatie care urmeaza imediat dupa numele programului in linia de comanda utilizata la executia acestuia. Pentru a primi acesti parametri se utilizeaza urmatoarele forme ale metodei Main( ):

```
public static void Main(string [ ] args)
public static int Main(string [ ] args)
```

Prima intoarce void iar cea de-a doua o valoare intreaga.

Exemplele 11 si 12. Metoda Main() cu parametri

```
using System;
class Descompunere
{
    public static void Main(string [] args)
    {
        int n;
        if (args.Length<1 )
      {
      Console.WriteLine("Introduceti un numar
      natural de la tastatura");
        n=int.Parse(Console.ReadLine());
        }
        else
        {
        n = int.Parse(args[0]);
        }
        int count = 2;
        Console.Write("{0}=", n);
        while (count <= n)
        {
            while (n % count == 0)
            {
                n = n / count;
                Console.Write("{0} ", count);
            }
            count++;
        }
    }
}
```

Rezultat: Programul descompune in factori primi un intreg introdus de la tastatura. Dupa caz, este utilizata metoda Main() cu sau fara parametri.

```
using System;

class TestDemo
{
    public static void Main(string [] args)
    {

Console.WriteLine("Sunt {0} parametrii ", args.Length);
 if (args.Length > 0)
  {
  Console.WriteLine("Acestia sunt:");
  }
for (int i = 0; i < args.Length; i++)
            Console.WriteLine(args[i]);
    }
}
```

Rezultat:

Programul returneaza parametrii introdusi in linia de comanda

Cuvantul cheie static

In mod obisnuit, un membru al unei clase trebuie sa fie accesat utilizand o instanta a acelei clase. Spre exemplu, in cel de-al doilea program din paragraful suparaincarcarea metodelor pentru a putea fi folosita metoda Aria( ) in blocul metodei Main( ) a fost creat un obiect ob de tipul clasei Overload.

Exista insa posibilitatea ca un membru al unei clase sa fie utilizat fara a depinde de vreo instanta. In acest caz, membrul respectiv este declarat ca static. Atat metodele cat si variabilele pot fi declarate statice.

Variabilele declarate ca static sunt variabile globale. La declararea unei instante a clasei, variabilele static nu sunt copiate. Ele sunt partajate de toate instantele clasei. Variabilele static sunt initializate la incarcarea clasei in memorie. Daca nu se specifica in mod explicit o valoare de initializare atunci o variabila static se initializeaza cu zero daca este tip numeric, cu null in cazul tipurilor de referinta si respectiv false pt tipul bool.

Diferenta dintre o metoda statica si o metoda obisnuita este ca metoda statica poate fi apelata fiind prefixata de numele casei din care face parte, fara crearea unei instante a clasei.

Exemplul 13. Initializarea si utilizarea variabilelor statice

```
using System;
class StaticD
{
    public static int a;
    public int b;
}
class StaticDemo
{
    public static void Main()
    {
        Console.WriteLine("Valoarea initiala a variabilei StaticD.a este {0}", StaticD.a);
        StaticD obj = new StaticD();
        obj.b = 10;
        //obj.a=20;  //fiind globala, variabila a poate fi initializata in acest mod
        StaticD.a = 20; //in schimb poate fi initializata manual sau printr-un constructor sau vreo metoda
        Console.WriteLine("Valoarea variabilei Static.a este {0}. Valoarea variabilei obj.b este {1}",
      StaticD.a, obj.b);
    }
}
```

Rezultat:

Valoarea initiala a variabilei StaticD.a este 0

Valoarea variabilei Static.a este 20. Valoarea variabilei obj.b este 10

Exemplul 14. Utilizarea metodelor statice

```
using System;
class Overload
{
   public static double Aria(double r)    {
      double A = 4 * (Math.Atan(1)) * r * r;
      Console.WriteLine("Aria cercului avand raza r={0} este A={1:#.##}", r, A);
      return A;
   }
   public static double Aria(double b, double h)   {
      double A;
      A = h * b / 2;
      Console.WriteLine("Aria triunghiului avand baza b={0} si inaltimea h={1} este A={2}", b, h, A);
      return A;
   }
   public static double Aria(double b, double B, double h)   {
      double A;
      A = (b + B) * h / 2;
      Console.WriteLine("Aria trapezului avand baza mica b={0}, baza mare B={1} si inaltimea h={2} este A={3}",
     b, B, h, A);
      return A;
   }
}
class DemoOverload
{
   public static void Main()    {
      Overload.Aria(3); Overload.Aria(2, 3); Overload.Aria(2, 3, 4);
   }
}
```

Rezultat:

Aria cercului avand raza r=3 este A=28,27
Aria triunghiului avand baza b=2 si inaltimea h=3 este A=3
Aria trapezului avand baza mica b=2, baza mare B=3 si inaltimea h=4 este A=10

Supraincarcarea operatorilor

Definitie. Procesul de redefinire a unui operator in contextul dat de o anumita clasa nou creata poarta numele de *supraincarcarea operatorilor*.

Observatii: -Supraincarcarea operatorilor este utila in numeroase aplicatii. Acest proces permite integrarea unei clase definite de utilizator in mediul de programare. Spre exemplu, daca o clasa defineste o lista de elemente atunci se poate utiliza operatorul + pentru adaugarea unui nou element in acea lista;

-La suparaincarcarea unui operator, semnificatia initiala a acestuia se conserva. Supraincarcarea reprezinta adaugarea unei noi operatii specifice unei anumite clase. Asadar, in cazul exemplului de mai sus, supraincarcarea lui + pentru adaugarea elementelor intr-o lista nu schimba semnificatia sa (care este de adunare) in contextul numerelor intregi;

- Pentru supraincarcarea unui operator se utilizeaza cuvantul cheie operator pentru a defini o metoda operator.

Formele generale pentru operatorii unari si respectiv opearatorii binari sunt:

```
public static tip-rez operator op(tip-param operand)
{ //operatii;}
 public static tip-rez operator op(tip-param1 operand, tip-param2 operand)
{ //operatii;}
```

unde: -operatorul supraincarcat +, -, etc. va substitui op;

-tip-rez reprezinta tipul valorii intoarse de operatia op. Valoarea intoarsa poate fi de orice tip, insa adeseori este de acelasi tip cu clasa pentru care are loc supraincarcarea operatorului;

-in cazul operatorului unar, operand trebuie sa fie de acelasi tip cu clasa pentru care se redefineste operatorul. In cazul operatorului binar, cel putin unul din operanzi (operand1 sau operand2) trebuie sa fie de acelasi tip cu clasa pentru care se redefineste operatorul.

-parametrii operatorilor nu pot utiliza modificatorii ref si out.

Exemplul 15. Supraincarcarea operatorilor binari

```
using System;
class Rlaen
{
    public int n;
    double[ ] vector;

    public Rlaen(int dimensiune)
    {
        n = dimensiune;
        vector=new double[n];
        for (int i = 0; i < vector.Length; i++)
            vector[i] = 0;
    }
    public Rlaen(int dimspatiu, params double[ ] v)
    {
        n = dimspatiu;
        vector = new double[n];
        for (int i = 0; i < vector.Length; i++)
            vector[i] = v[i];
    }
    public static Rlaen operator +(Rlaen op1, Rlaen op2)
    {
        if (op1.n == op2.n)
        {
            Rlaen rezultat = new Rlaen(op1.n);
            for (int i = 0; i < op1.vector.Length; i++)
            {
                rezultat.vector[i] = op1.vector[i] + op2.vector[i];
            }
            return rezultat;
        }
        else
        {
            Console.WriteLine("Cei doi operanzi nu au aceeasi
        dimensiune. Operatia nu se poate realiza");
            Rlaen rezultat = new Rlaen(op1.n);
            return rezultat;
        }
    }
public static Rlaen operator -(Rlaen op1, Rlaen op2)
    {

        if (op1.n == op2.n)
        {
            Rlaen rezultat = new Rlaen(op1.n);
            for (int i = 0; i < op1.vector.Length; i++)
            {  rezultat.vector[i] = op1.vector[i] - op2.vector[i];  }
            return rezultat;
        }
        else
        {
            Console.WriteLine("Cei doi operanzi nu au aceeasi dimensiune.
        Operatia nu se poate realiza");
            Rlaen rezultat = new Rlaen(op1.n);
            return rezultat;
        }
    }
    public void show()
    {
        Console.Write("(");
        for (int i=0; i<vector.Length-1; i++)
        Console.Write("{0:#.##},",vector[i]);
        Console.Write("{0:#.##})", vector[vector.Length-1]);
        Console.WriteLine();
    }
}
class RlaenDemo
{
    public static void Main()
    {
        Rlaen a = new Rlaen(4, 1, 2, 3, 4);
        Rlaen b = new Rlaen(4, 4, 3,2,1);
        Rlaen c = new Rlaen(4);
        Console.Write("a=");     a.show();
        Console.Write("b=");     b.show();
        c = a + b;   Console.Write("a+b=");   c.show();
        c = a - b;   Console.Write("a-b=");  c.show();
    }
}
```

In exemplul anterior, au fost supraincarcati operatorii binari + si – pentru o clasa care reprezinta din punct de vedere matematic spatiul R^n. Cei doi operanzi au acelasi tip.

Exista posibilitatea de a supraincarca operatori pentru care tipurile operanzilor sunt diferite. Spre exemplu, in clasa Rlaen din exemplul anterior putem considera metodele * (alaturate) pentru care unul din operanzi este de tip double, iar celalalt este un obiect din clasa Rlaen. Din punct de vedere matematic, am introdus cele doua operatii care organizeaza spatiul R^n ca un spatiu vectorial peste R, operatia interna de adunare si operatia externa de inmultire cu scalari.

Un aspect interesant este acela ca o operatie redefinita, poate la randul ei sa fie suparaincarcata. Este cazul operatorului * care este definit in doua moduri.Din punct de vedere matematic oricare dintre cele doua metode conduce la acelasi rezultat. Bineinteles aceasta nu este o regula.

Toate metodele utilizate in exemplul anterior au creat in blocul lor un obiect instanta a clasei Rlaen.

```
public static Rlaen operator *(double l, Rlaen op2)
   {
         Rlaen rezultat = new Rlaen(op2.n);
         for (int i = 0; i < op2.vector.Length; i++)
         {
            rezultat.vector[i] = l* op2.vector[i];
         }
         return rezultat;



   }
   public static Rlaen operator *(Rlaen op2, double l)
   {
      Rlaen rezultat = new Rlaen(op2.n);
      for (int i = 0; i < op2.vector.Length; i++)
      {
         rezultat.vector[i] = l * op2.vector[i];
      }
      return rezultat;
   }
```

Supraincarcarea operatorilor unari

In cazul operatorilor unari, singurul operand trebuie sa fie acelasi tip cu clasa pentru care se redefineste operatorul, in cazul nostru un obiect instanta a clasei Rlaen.

Alaturat sunt exemplificate trei metode. Prima dintre ele nu modifica operandul, intrucat in blocul metodei este creat un nou obiect care este intors ca rezultat. In schimb, celelalte doua metode modifica parametrul op1.

```
public static Rlaen operator -(Rlaen op1)
{

    Rlaen rezultat = new Rlaen(op1.n);
    for (int i = 0; i < op1.vector.Length; i++)
    { rezultat.vector[i] = -op1.vector[i]; }
    return rezultat;
}

public static Rlaen operator ++(Rlaen op1)
{

    for (int i = 0; i < op1.vector.Length; i++)
    { op1.vector[i] = op1.vector[i]+1; }
    return op1;
}

public static Rlaen operator --(Rlaen op1)
{

    for (int i = 0; i < op1.vector.Length; i++)
    { op1.vector[i] = op1.vector[i] - 1; }
    return op1;
}
```

Supraincarcarea operatorilor relationali

Operatorii relationali, cum ar fi ==, <, <=, pot de asemenea fi supraincarcati. De regula, un operator relational supraincarcat va intoarce o valoare true sau false, desi nu este neaparat. Insa daca alegeti sa intoarceti alt tip se pot crea confuzii.

Exista o restrictie importanta referioare la redefinirea operatorilor relationali si anume aceea ca trebuie redefiniti in perechi. Adica, daca este redefinit operatorul < atunci trebuie redefinit si operatorul >. Perechile sunt urmatoarele: (==, !=); (<,>); (<=, >=).

Alaturat sunt redefiniti operatorii == si != pectru clasa Rlaen.

Exista cativa operatori (altii decat cei relationali) care nu pot fi supraincarcati, si anume: &&, ||, [ ], ( ), new, is, sizeof, ?,. =, +=, -=.

Cuvintele cheie true sau false pot fi utilizate ca operatori unari in scopul redefinirii.

```
public static bool operator ==(Rlaen op1, Rlaen op2)
 {    if (op1.n == op2.n)
      {    bool[ ] compar = new bool[op1.vector.Length];
           for (int i = 0; i < op1.vector.Length; i++)
           {    compar[i]=(op1.vector[i]== op2.vector[i]);
                 if (!compar[i])
                {  return false;   break; }
           }

         return true;
      }
      else
      {   Console.WriteLine("Cei doi operanzi nu au aceeasi
      dimensiune. Operatia nu se poate realiza");
         return false; }
   }
```

```
public static bool operator !=(Rlaen op1, Rlaen op2)
{   if (op1.n == op2.n)
   {   bool[] compar = new bool[op1.vector.Length];
      for (int i = 0; i < op1.vector.Length; i++)
      {   compar[i] = (op1.vector[i] != op2.vector[i]);
         if (compar[i])
         { return true;   break;  }
      }
      return false;
   }
   else
   {   Console.WriteLine("Cei doi operanzi nu au aceeasi
   dimensiune. Operatia nu se poate realiza");
      return false;   }
}
```

Indexari, proprietati, structuri, enumerari

Indexari

O indexare permite unui *obiect sa fie indexat la fel ca un tablou*. In principal, indexarile se utilizeaza la crearea *de tablouri specializate care sunt obiectul unor restrictii*. Indexarile pot avea una sau mai multe dimensiuni.

In cazul indexarilor unidimensionale, forma generala este:

```
tip-element this[int index]
{
get {  //se intoarce valoarea precizata de index }
set  {  //se modifica valoarea precizata de index}
}
```

unde: - tip-element reprezinta tipul de baza al indexarii. Fiecare element al indexarii este de tipul tip-element;

-paramentrul index primeste valoarea indexului elementului care va fi accesat; Acest parametru poate fi si de alt tip decat int, insa intrucata indexarile se utilizeaza pentru a implementa un mecanism similar tablourilor, de regula se recurge la un tip intreg.

-in corpul unei indexari sunt definiti doi accesori, denumiti get si set. Un accesor este similar cu o metoda, cu diferenta ca nu contine parametrii si nu specifica tipul rezultatului. La utilizarea indexarii, accesorii sunt automat apelati, ambii accesori primind index ca parametru: Daca indexarea apare in partea stanga a unei instructiuni de atribuire, se apeleaza accesorul set. Acesta primeste, in plus fata de parametrul index, o valoare numita value, care este atribuita elementului indexarii cu indexul precizat de parametrul index. Daca indexarea apare in partea dreapta a unei expresii atunci se apeleaza accesorul get, care intoarce valoarea asociata cu index.

Unul dintre avantajele utilizarii unei indexari este acela ca se *poate controla cu exactitate modul de acces la un tablou*, eliminand accesarile incorecte (utilizarea unor indici negativi sau indici mai mari decat dimensiunea tabloului).

Exemplul 1. Utilizarea indexarilor

```
using System;
class Tablou
{
   int[] a;
   public int dimensiune;
   public bool coderr;
   public Tablou(int lungime)
   {
      a = new int[lungime];
      dimensiune= lungime;
   }
   public int this[int index]
   {
      get
      {
         if (ok(index))
         {
            coderr = false;
            return a[index];
         }
         else
         { coderr = true;  return -1;  }
      }
      set
      {
         if (ok(index))
         {
            a[index] = value;
            coderr = false;
         }
         else {  coderr = true;}
      }
   }
private bool ok(int index)
   {
      if((index>=0) & (index<dimensiune))
         return true;
      else
         return false;
   }
}
class TablouDemo
{
   public static void Main()
   {
      int x;
      Tablou tb = new Tablou(5);
      for (int i = 0; i < tb.dimensiune*2; i++)
         tb[i] = 10 * i;
      for (int i = 0; i < tb.dimensiune*2; i++)
      {
         x = tb[i];
         if (tb.coderr)
            Console.Write("\ntb[{0}] Depaseste 
     marginile", i);
         else
            Console.Write("{0} \t",x);
      }
   }
}
```

Rezultat:

0 10 20 30 40

tb[5] Depaseste marginile
tb[6] Depaseste marginile
tb[7] Depaseste marginile
tb[8] Depaseste marginile
tb[9] Depaseste marginile

Exemplul 2. Utilizarea indexarilor

Nu este neaparat ca o indexare sa lucreze cu un tablou. Indexarile ofera functionalitate care se apropie de cea a tablourilor. Spre exemplu, programul alaturat contine o indexare care se comporta ca un tablou accesibil la citire si care contine puterile lui 3.

De asemenea, nu este neaparat ca prametrul index sa fie de tip int. In acest sens, realizati in programul alaturat urmatoarele modificari pentru a obtinte puterile lui 3 la o putere de tip double: schimbati in tot corpul programului int cu double, stergeti castul (int) din accesorul get si initializati variabila i din instructiunea for cu o valoare rationala, spre exemplu 0.1.

In cazul tablourilor bidimensionale, o indexare are forma:

```
tip-element this[int index1, int index2]
{
get {  //se intoarce valoarea precizata de index1 si index2 }
set  {  //se modifica valoarea precizata de index1 si index2}
}
```

Exercitiu: Modificati exemplul 1 in asa fel incat sa implementati o indexare bidimensionala.

```
using System;
class Putere3
{
    public int this[int index]
    {
        get
        {
            if (index >= 0)
                return (int)Math.Pow(3, index);
            else
                return -1;
        }
    }
}
class PutereDemo
{
    public static void Main()
    {
        Putere3 p = new Putere3();
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine("3^{0}={1}", i, p[i]);
        }
    }
}
```

Rezultat:

```
3^0=1
3^1=3
3^2=9
3^3=27
3^4=81
3^5=243
3^6=729
3^7=2187
3^8=6561
3^9=19683
```

Proprietati

Unul din beneficiile programelor orientate obiect il reprezinta capacitatea de a *controla reprezentarea interna si accesul la date*. In acest sens, C# furnizeaza un concept (implementeaza o categorie de membrii ai claselor) numit *proprietati*. O *proprietate* gestioneaza accesul la un camp (o variabila privata) prin intermediul accesorilor get si set.

Forma generala a unei proprietati este:

```
tip nume
{
get {  //codul accesorului get }
set  {  //codul accesorului set}
}
```

unde tip precizeaza tipul proprietatii, ca de exemplu int, iar nume este numele proprietatii. La fel ca in cazul indexarilor, accesorul set primeste un parametru numit value, care contine valoarea atribuita propretatii.

Observatii:-este esential sa intelegem ca proprietatile nu definesc locatii de memorie. Gestionand accesul la un camp, acesta din urma trebuie specificat separat de proprietate. Proprietatea nu contine campul respectiv;

-se pot defini proprietati accesibile la citire si scriere (contin ambii accesori get si set) sau accesibile doar la citire (contin accesorul get) sau la scriere (contin accesorul set);

-deoarece proprietatile nu definesc o zona de memorie, acestea nu pot fi transmise ca parametri ref sau out unei metode;

-proprietatile nu pot fi supraincarcate;

-o proprietate nu trebuie sa altereze starea variabilei asociate la apelul accesorului get.

In exemplul urmator se considera utilizeaza doua proprietati pentru a accesa la citire si scriere doua campuri private my_x si my_y.

Exemplul 3. Proprietati

```
using System;
class Point
{
   double my_x;
   double my_y;
   public double x
   {
      get { return my_x; }
      set { if (value<=0) my_x = value;      else my_x=-value;  }
   }
   public double y
   {
      get { return my_y; }
      set {  if (value <= 0) my_y = value;     else my_y = -value; }
   }
}
class Segmdr
{
   public static void Main()
   {
      Point punct1 = new Point();  Point punct2 = new Point();
      double dist;
      punct1.x = 3;      punct1.y = 4;
      punct2.x = 5;      punct2.y = 3;
      dist = Math.Sqrt((punct1.x - punct2.x) * (punct1.x - punct2.x) + (punct1.y - punct2.y) * (punct1.y - punct2.y));
      Console.WriteLine("Distanta dintre punctele ({0},{1}) si ({2},{3}) este: {4:#.##}", punct1.x, punct1.y, punct2.x,
     punct2.y, dist);
   }
}
```

Rezultat:

Distanta dintre punctele (-3,-4) si (-5,-3) este: 2,24

Structuri

Clasele sunt tipuri de referinta. In consecinta obiectele claselor sunt accesate prin intermediul referintelor, spre deosebire de tipurile valorice care sunt accesate direct. Exista insa cazuri cand este util sa accesati un obiect in mod direct. Unul din motive o reprezinta eficienta.

Pentru a eficientiza programele, C# pune la dispozitie structurile. O structura este similara din punct de vedere sintactic unei clase, fiind insa de tip valoric si nu de referinta.

Forma generala a unei structuri este:

```
struct nume
{ //declaratii de membrii}
```

unde nume reprezinta numele structurii, iar struct reprezinta cuvantul cheie care declara structura.

Observatii: -ca si in cazul claselor, membrii structurilor pot fi *metode, variabile, indexari, proprietati, metode operator supraincarcate, evenimente*. De asemenea, se pot defini *constructori insa nu destructori*. In ceea ce priveste constructorii, *nu se pot defini constructori impliciti (fara parametri)* pentru structuri. Aceasta datorita faptului ca un constructor implicit este definit automat pentru toate structurile si acesta nu poate fi modificat;

-spre deosebire de clase, *structurile nu pot mosteni alte structuri sau alte clase* si nu pot fi utilizate ca baze pentru alte structuri sau clase. Ele pot implementa insa interfete;

-un obiect structura poate fi creat utilizand operatorul new in acelasi mod ca si pentru o instanta a unei clase, insa nu este obligatoriu. Daca nu se utilizeaza operatorul new atunci obiectul este creat fara a fi initializat. Initializarea trebuie facuta manual;

-structurile din C# difera din punct de vedere conceptual fata de structurile din C++. In C++ struct si class sunt aproape echivalente. Diferenta consta in accesul implicit la membrii, care este privat pentru class si public pentru struct. In C#, struct defineste un tip valoric, in timp ce class un tip de referinta.

Exemplul 4. Structuri

```
using System;
struct Cont {
   public string nume;
   public decimal sold;
   public Cont(string n, decimal s)   {   nume = n;   sold = s;  }
}
 class StructDemo{
   public static void Main()  {
      Cont c1 = new Cont("Gheorghe", 1244.926m);
      Cont c2 = new Cont();
      Cont c3;
      Console.WriteLine("{0} are un sold de {1:#.##}", c1.nume,c1.sold);
      Console.WriteLine();
      if (c2.nume == null)
      {
         Console.WriteLine("c2.nume este null");
         Console.WriteLine("c2.sold este {0}", c2.sold);
         Console.WriteLine();
      }
      c3.nume = "Ana";
      c3.sold = 100000m;
      Console.WriteLine("{0} are un sold de {1:C}", c3.nume, c3.sold);
      Console.WriteLine();
   }
}
```

Rezultat:
Gheorghe are un sold de 1.244,93 lei

c2.nume este null
c2.sold este 0

Ana are un sold de 100.000,00 lei

Enumerari

O *enumerare* este o multime de <u>constante intregi cu nume</u>, care specifica toate valorile posibile pe care le pot lua variabilele care au ca tip enumerarea.

Enumerarile sunt des intalnite in viata de zi cu zi. Spre exemplu, o enumerare a bancnotelor utilizate in Romania (1 leu, 5 lei, 10 lei, 50 lei, 100 lei, 200 lei, 500 lei) sau o enumerare a zilelor saptamanii (Luni, Marti etc.)

Un tip enumerare se declara utilizand cuvantul cheie enum. Forma generala a unei enumerari este:

```
acces enum nume {lista-constante};
```

unde numele tipului enumerare este specificat prin nume, iar lista-constante este o lista de identificatori separati prin virgule. Spre exemplu, secventa urmatoare de cod defineste o enumerare numita bancnote:

```
enum bancnote {unLeu, cinciLei, zeceLei, cincizeciLei, osutaLei, douasuteLei, cincisuteLei};
```

Observatii:-aspectul esential care trebuie accentuat este ca fiecare constanta dintr-o enumerare reprezinta o valoare intreaga. Fiecare constanta simbolica primeste o valoare mai mare cu o unitate fata de constanta precedenta din enumerare. In mod implicit, valoarea primei constante este 0. In cazul exemplului de mai sus constanta unLeu are valoarea 0, constanta cinciLei are valoarea 1, etc

-membrii unei enumerari pot fi accesati prin intermediul numelui tipului enumerarii cu ajutorul operatorului punct (.). Astfel, intructiunile: bancnote bancnotaMea=bancnote.zeceLei;
Console.WriteLine(“{0}:{1}", bancnotaMea, (int)bancnotaMea); afiseaza zeceLei: 2.

-se poate preciza in mod explicit valoarea uneia sau mai multor constante simbolice, ca in exemplul de mai jos. In acest caz constantele simbolice care urmeaza vor primi valori mai mari decat cele situate inaintea lor:

```
enum bancnote {unLeu, cinciLei, zeceLei, cincizeciLei=50, osutaLei, douasuteLei, cincisuteLei};
```

Valorile constantelor sunt acum: unLeu 0, cinciLei 1, zeceLei 2, cincizeciLei 50, osutaLei 51, douasuteLei 52, cincisuteLei 53

-in mod implicit, enumerarile au tipul int. Puteti insa sa creati o enumerare de orice tip intreg, cu exceptia lui char. Pentru a specifica un alt tip de baza decat int, tipul trebuie precizat dupa numele enumerarii, fiind separat de acesta prin doua puncte (ca in exemplul de mai jos).

```
enum bacnote: byte {unLeu, cinciLei, zeceLei, cincizeciLei, osutaLei, douasuteLei, cincisuteLei};
```

Exemplul 5. Enumerari

```
using System;
public class Bday
{
enum Month
{   January = 1, February,  March,  April, May, June, July , August, September, October, November,
      December }

 struct birthday
 {
    public Month bmonth;
    public int bday;
    public int byear;
 }

 public static void Main()
 {
     birthday MyBirthday;

     MyBirthday.bmonth = Month.August;
     MyBirthday.bday = 11;
     MyBirthday.byear = 1955;

    System.Console.WriteLine("My birthday is {0} {1}, {2}",
    MyBirthday.bmonth, MyBirthday.bday, MyBirthday.byear);
 }
 }
```

Rezultat:

My birthday is August 11, 1955

Notiuni despre mostenire

Introducere

Mostenirea, alaturi de polimorfism, incapsulare si reutilizare a codului reprezinta principiile fundamentale ale programarii orientate obiect.

Utilizand mostenirea, se poate crea o clasa generala, care defineste caracteristici comune unei multimi si care poate fi mostenita de alte clase, mai specifice, fiecare adaugand numai caracteristicile care o identifica in mod unic.

O clasa care este mostenita se numeste *clasa de baza*, iar clasa care mosteneste anumite caracteristici se numeste *clasa derivata*.

Forma generala a unei declaratii class care mosteneste o clasa de baza este:

```
class nume-clasa-derivata : nume-clasa-baza
{
   //corpul clasei derivate
}
```

Spre deosebire de C++, in C# orice clasa derivata pe care o creati poate mosteni doar o singura clasa de baza. Se poate insa sa creati o ierarhie de clase, in care o clasa derivata sa devina clasa de baza din care sunt derivate alte clase.

In exemplul urmator este creata o clasa de baza numita Persoana care memoreaza numele, locul nasterii, data nasterii, dupa care creeaza o clasa derivata numita Student care adauga campul facultatea si metoda showStud().

Exemplul 1. Mostenire

```
using System;
class Persoana
{
   public string nume;   public string locnas;    public string datanas;
   public void showPers()
   {   Console.WriteLine("{0} s-a nascut in {1} la data de {2}",nume, locnas, datanas);    }
}
class Student:Persoana
{
   public string facultatea;
   public void showStud()
   {    Console.WriteLine(“Este student(a) la Facultatea de {0}",facultatea);   }
}

class PersoanaDemo
{
   public static void Main()
   {
      Student s1 = new Student();
      Student s2 = new Student();
      s1.nume = "Popescu Vasile";   s1.locnas = "Suceava";  s1.datanas = "23 iunie 1985";
      s1.facultatea = "Matematica";   s1.showPers();   s1.showStud();
      Console.WriteLine();
      s1.nume = "Ionescu Ioana";      s1.locnas = "Vaslui";     s1.datanas = "1 martie 1986";
      s1.facultatea = "Fizica";      s1.showPers();      s1.showStud();
   }
}
```

Rezultat:

Popescu Vasile s-a nascut in Suceava la data de 23 iunie 1985

Este student(a) la Facultatea de Matematica

Ionescu Ioana s-a nascut in Vaslui la data de 1 martie 1986

Este student(a) la Facultatea de Fizica

Accesul la membrii

Adeseori membrii unei clase sunt declarati ca privati pentru a evita utilizarea neautorizata a acestora. Mostenirea unei clase nu anuleaza restrictiile impuse de accesul *privat* (membrii privati ai unei clase nu pot fi accesati de clasele derivate).

Pe de alta parte, interzicerea accesului la membrii privati este de multe ori prea severa. Pentru a depasi acest inconvenient, se pot utiliza proprietatile publice care sa asigure accesul la membrii privati sau in cazul mostenirii utilizarea membrilor *protected*. Un membru protected este acesibil din clasa data si din clasele derivate.

Prezentam o varianta a exemplului anterior in care o parte din membrii sunt implementati cu ajutorul proprietatilor, iar altii sunt protejati. *Observati ca au fost introdusi constructori, aceasta deoarece membrii protected nu pot fi initializati manual din afara clasei de baza si clasei derivate.*

Exemplul 2. Membrii protejati

```
using System;
class Persoana
{
    protected string nume;    protected string locnas;
    private string my_datanas;
    public string datanas    {
        get{return my_datanas;}
        set { my_datanas = value; }
    }
public Persoana(string n, string l, string d)    {
        nume = n; locnas = l; datanas = d;
    }
    public void showPers()
    { Console.WriteLine("{0} s-a nascut in {1} la data de
      {2}", nume, locnas, datanas); }
}
```

```
class Student : Persoana
{
    string my_facultatea;
    public string facultatea
    {
        get{return my_facultatea;}
        set { my_facultatea = value; }
    }
    public Student(string n, string l, string d, string f)
        : base(n, l, d)
    {
        facultatea=f;
    }

    public void showStud()
    { Console.WriteLine("Student(a) la Facultatea de {0}",
      facultatea); }
}

class PersoanaDemo
{
    public static void Main()
    {
        Student s1 = new Student("Popescu Ion", "Bacau",
      "22 mai 1984", "Matematica");
         s1.showPers(); s1.showStud();
    }
}
```

Rezultat:

Popescu Ion s-a nascut in Bacau la data de 22 mai 1984
Este student(a) la Facultatea de Matematica

Constructorii

Intr-o ierarhie, atat clasele de baza cat si cele derivate pot dispune de constructori proprii. Pentru construirea unui obiect apartinand unei clase derivate se utilizeaza ambii constructori (atat cel din clasa de baza cat si cel din clasa derivata). Constructorul clasei de baza construieste portiunea obiectului care apartine de clasa de baza, iar constructorul clasei derivate construieste portiunea care apartine de clasa derivata.

In practica se procedeaza astfel:

- atunci cand ambele clase nu dispun de constructori se utilizeaza constructorii impliciti;

-atunci cand doar clasa derivata defineste un constructor, obiectul clasei derivate se construieste utilizand pentru partea care corespunde clasei de baza constructorul implicit;

-atunci cand cand ambele clase dispun de constructori, se poate proceda in doua moduri. Fie se utilizeaza o clauza base (ca in exemplul anterior), apelandu-se constructorul clasei de baza pentru construirea partii obiectului care tine de clasa de baza. Fie nu este utilizata o clauza base, iar atunci partea care apartine clasei de baza se construieste cu ajutorul constructorului implicit.

In exemplul anterior a fost utilizata o clauza base pentru a utiliza facilitatile oferite de constructorul clasei de baza.

Ascunderea numelor

Este posibil ca o clasa derivata sa defineasca un membru al carui nume sa coincida cu numele unui membru din clasa de baza. In acest caz, membrul clasei de baza nu va fi vizibil in clasa derivata.

Din punct de vedere tehnic aceasta nu este o eroare, insa compilatorul va genera un mesaj de avertisment care informeaza utilizatorul ca un nume a fost ascuns. Daca intradevar se doreste ascunderea unui membru al clasei de baza atunci se poate utiliza cuvantul cheie new (ca in exemplul de mai jos) pentru ca mesajul de avertizare sa nu mai apara.

Exemplul 3. Ascunderea numelor

```
using System;
class A     {public int i = 1;}
class B : A
{ public new int i;
public B(int b)  {i = b;}
}
class DemoAB
{
 public static void Main()
{ B ob = new B(10);
Console.WriteLine("i={0}", ob.i);}
}
```

Rezultat:

i=10

Cuvantul cheie base poate fi utilizat si in alt scop decat cel utilizat in paragraful anterior. Se comporta oarecum similar cu this, referindu-se la clasa de baza a clasei derivate pentru care se utilizeaza. Aceasta versiune are forma generala: base.membru; unde membru poate desemna atat o metoda cat si o variabila. Cuvatul cheie base permite accesul la variabila sau metoda ascunsa.

Exemplul 4. Ascunderea numelor si utilizarea clauzei base

```
using System;
class A
{public int i = 1;
public void show()
{ Console.WriteLine("In clasa de baza i={0}", i); }
}
class B : A
{ new public int i;
public B(int a, int b)
{ base.i = a; i = b; }
new public void show()
{   base.show();
Console.WriteLine("In clasa derivata i={0}", i); }
}
class DemoAB
{
public static void Main()
{  B ob = new B(5, 10);
ob.show(); }
}
```

Rezultat:

In clasa de baza i=5

In clasa derivata i=10

Ierarhii pe mai multe nivele

Pana acum, am creat ierarhii simple care contin numai o clasa de baza si o clasa derivata. Se pot insa construi ierarhii care contin oricate niveluri de mostenire doriti. Utilizarea unei clase derivate pe post de clasa de baza pentru o alta clasa este corecta.

Mai jos, este utilizata clasa derivata Student pe post de clasa de baza pentru a creia clasa derivata StudentMath.

Un alt aspect important: base refera intodeauna constructorul din clasa de baza cea mai apropiata.

De asemenea, intr-o ierarhie de clase, constructorii sunt apelati in ordinea derivarii: de la clasa de baza la clasele derivate.

Exemplul 5. Ierarhii pe doua nivele

.

```
using System;
class Persoana {
    protected string nume;
    protected string locnas;
    private string my_datanas;
    public string datanas    {
        get{return my_datanas;}
        set { my_datanas = value; }
    }
    public Persoana(string n, string l, string d)    {
        nume = n; locnas = l; datanas = d;
    }
    public void showPers()    {
    Console.WriteLine("{0} s-a nascut in {1} la data de
      {2}", nume, locnas, datanas); }
}
class Student : Persoana
{   string my_facultatea;
    public string facultatea
    {   get { return my_facultatea; }
        set { my_facultatea = value; }
    }
    public Student(string n, string l, string d, string f) : base(n, l,
        d)
    {  facultatea = f; }
 }
class StudentMath : Student
{    string my_specializarea;
    public string specializarea
    {   get { return my_specializarea; }
        set { my_specializarea = value; }
    }
    public StudentMath(string n, string l, string d, string f, string
        s)
        : base(n, l, d,f)
    {    specializarea = s;   }
     public void showStudMath()
    { Console.WriteLine("Student(a) la Facultatea de {0},
        specializarea {1}", facultatea, specializarea); }
}
class PersoanaDemo{
    public static void Main()  {
        StudentMath s1 = new StudentMath("Popescu Ion",
        "Bacau", "22 august 1984", "Matematica", "Matematica-
        Informatica");
        s1.showPers(); s1.showStudMath();
    }
}
```

Rezultat:

Popescu Ion s-a nascut in Bacau la data de 22 august 1984

Student(a) la Facultatea de Matematica, specializarea Matematica-Informatica

Variabilelor de tipul clasei de baza le pot fi atribuite instante ale claselor derivate

Limbajul C# este puternic tipizat. In afara conversiilor standard si a promovarilor automate care se aplica tipurilor simple, *compatibilitatea tipurilor este strict controlata*. Aceasta inseamna ca o variabila referinta de tipul unei clase nu poate, in majoritatea cazurilor, sa refere un obiect apatinand unei alte clase.

Exista insa o exceptie. O variabila referinta de tipul unei clase de baza poate sa refere un obiect apartinand oricarei clase derivate din clasa de baza.

In acest *caz tipul variabilei referinta si nu obiectul la care acesta se refera determina membrii care pot fi accesati*. In consecinta, cand se atribuie o referinta catre o instanta a clasei derivate unei variabile referinta de tipul clasei de baza, veti avea acces numai la partile obiectului definite in clasa de baza. In programul urmator (Exemplul 6), x2 nu are acces la variabila b.

Din punct de vedere practic, exista doua cazuri importante cand se atribuie referinte catre clase derivate unor variabile de tipul clasei de baza. Un prim caz este la apelul constructorilor intr-o ierarhie de clase (vezi exemplul 7, unde constructorii accepta ca parametrii instante ale claselor lor). Celalalt caz se refera la metode virtuale.

Exemplul 6. Atribuirea unor instante ale claselor derivate variabilelor de tipul clasei de baza

```
using System;
class A
{
   public int a;
   public A(int i) { a = i; }
}
class B : A
{
   public int b;
   public B(int i, int j)   : base(i)
   {
      b = j;
   }
}
class RefBaza
{
   public static void Main()
   {
      A x = new A(10);
      A x2;
      B y = new B(3, 4);
      x2 = x;                   //corect, fiind de acelasi tip
      Console.WriteLine("x2.a={0}", x2.a);
      x2 = y;                  //corect deoarece B deriva din A
      Console.WriteLine("x2.a={0}",x2.a);
      //Console.WriteLine("x2.b={0}", x2.b); //eroare clasa A nu contine membrul b
    }
}
```

Rezultat:

x2.a=10

x2.a=3

Exemplul 7. In programul de mai jos constructorul clasei Persoana (referit prin clauza base) asteapta un obiect Persoana, insa Student() ii transmite un obiect de tipul Student. Motivul pentru care se poate proceda astfel este ca o referinta de tipul clasei de baza poate referi o instanta a unei clase derivate.

```
using System;
class Persoana
{
   protected string nume;
   protected string locnas;
   private string my_datanas;
   public string datanas
   {
      get{return my_datanas;}
      set { my_datanas = value; }
   }
   public Persoana(string n, string l, string d)
   {
      nume = n; locnas = l; datanas = d;
   }
   public Persoana(Persoana ob)
   {
      nume = ob.nume; locnas = ob.locnas; datanas =
      ob.datanas;
   }
   public void showPers()
   { Console.WriteLine("{0} s-a nascut in {1} la data de
      {2}", nume, locnas, datanas); }
}
class Student : Persoana
{   string my_facultatea;
   public string facultatea
   {   get { return my_facultatea; }
      set { my_facultatea = value; }
   }
   public Student(string n, string l, string d, string f)
     : base(n, l, d)
   {  facultatea = f;  }
   public Student(Student ob)
      : base(ob)
   { facultatea = ob.facultatea;  }
   public void showStud()
   { Console.WriteLine("Student(a) la Facultatea de {0}",
      facultatea); }
}
class PersoanaDemo
{  public static void Main()
   {   Student s1 = new Student("Popescu Ion", "Bacau",
      "22 august 1984", "Matematica");
      Student s2 = new Student(s1);
      Console.WriteLine("Informatii despre s1:");
      s1.showPers(); s1.showStud();
      Console.WriteLine();
      Console.WriteLine("Informatii despre s2:");
      s1.showPers(); s1.showStud();
   }
}
```

Rezultat:

Informatii despre s1:
Popescu Ion s-a nascut in Bacau la data de 22 august 1984
Student(a) la Facultatea de Matematica

Informatii despre s2:
Popescu Ion s-a nascut in Bacau la data de 22 august 1984
Student(a) la Facultatea de Matematica

Metode virtuale

O metoda virtuala este o metoda declarata ca virtual intr-o clasa de baza si redefinita apoi in una din clasele derivate. Fiecare din clasele derivate pot dispune de o versiune proprie a unei metode virtuale.

Metodele virtuale sunt interesante atunci cand sunt apelate prin intermediul unei referinte de tipul clasei de baza. In acest caz compilatorul determina versiunea care va fi apelata tinand cont de tipul obiectului referit de referinta si nu de tipul referintei. Determinarea se efectueaza la momentul executiei. Cu alte cuvinte tipul obiectului (si nu tipul referintei) determina care versiune a metodei virtuale se va executa. Astfel daca o clasa de baza contine o metoda virtuala si exista clase derivate din clasa de baza atunci cand se refera tipuri diferite de obiecte prin intermediul unei referinte de tipul clasei de baza, se vor executa versiuni diferite ale metodei virtuale.

Declararea unei metode virtuale in clasa de baza se realizeaza cu cuvantul cheie virtual. La redefinirea metodei in cadrul claselor derivate se utilizeaza modificatorul override. Acest proces de redefinire a unei metode se numeste extindere a metodei. La extinderea unei metode numele si signatura metodei extinse trebuie sa coincida cu numele si signatura metodei virtuale care se extinde. Metodele virtuale nu pot fi declarate ca static sau abstract.

Extinderea metodelor reprezinta baza pentru unul dintre cele mai puternice concepte implementate in C#: apelarea dinamica a metodelor. Apelarea dinamica a metodelor este mecanismul prin care rezolvarea apelului unei metode extinse este amanata de la momentul compilarii pana la momentul executiei programului. Apelarea dinamica a metodelor este modalitatea prin care limbajul C# implementeaza polimorfismul la executie.

Exemplul 8. Tipul obiectului referit (si nu tipul referintei) este cel care stabileste ce versiune a metodei virtuale se va executa.

```
using System;
class Baza
{    public virtual void Care()
   {  Console.WriteLine("Metoda Care() din clasa de baza"); }
}
class Derivata1:Baza
{    public override void Care()
  {  Console.WriteLine("Metoda Care() din clasa Derivata1"); }
}
class Derivata2 : Derivata1
{   public override void Care()
   {  Console.WriteLine("Metoda Care() din clasa Derivata2");
      }
}
class ExtindereDemo
{
   public static void Main()
   {
      Baza obB = new Baza();
      Derivata1 obD1 = new Derivata1();
      Derivata2 obD2 = new Derivata2();
      Baza a;
      a = obB;   a.Care();
      a = obD1;    a.Care();
      a = obD2;     a.Care();
   }
}
```

Rezultat:

Metoda Care() din clasa de baza
Metoda Care() din clasa Derivata1
Metoda Care() din clasa Derivata2

Exemplul 9. Nu este obligatoriu ca o metoda virtuala sa fie extinsa. Daca o clasa derivata nu contine o versiune proprie a unei metode virtuale atunci se utilizeaza versiunea din clasa de baza.

```
using System;
class Baza
{    public virtual void Care()
   { Console.WriteLine("Metoda Care() din clasa de
      baza"); } }
class Derivata1 : Baza  {   }
class Derivata2 : Derivata1
{    public override void Care()
   { Console.WriteLine("Metoda Care() din clasa
      Derivata2"); } }
class ExtindereDemo
{
   public static void Main()
   {
      Baza obB = new Baza();
      Derivata1 obD1 = new Derivata1();
      Derivata2 obD2 = new Derivata2();
      Baza a;
      a = obB; a.Care();
      a = obD1; a.Care();
      a = obD2; a.Care();
   }
}
```

Rezultat:

Metoda Care() din clasa de baza
Metoda Care() din clasa de baza
Metoda Care() din clasa Derivata2

Exemplul 10. Extinerea metodei show() din cadrul exemplelor anterioare

```
using System;
class Persoana
{
   protected string nume;
   protected string locnas;
   private string my_datanas;
   public string datanas
   {
      get{return my_datanas;}
      set { my_datanas = value; }
   }
   public Persoana(string n, string l, string d)
   {
      nume = n; locnas = l; datanas = d;
   }
   public virtual void show()
   { Console.WriteLine("{0} s-a nascut in {1} la data de {2}", nume, locnas,
        datanas); }
}
class Student : Persoana
{
   string my_facultatea;
   public string facultatea
   {
      get { return my_facultatea; }
      set { my_facultatea = value; }
   }
   public Student(string n, string l, string d, string f)
      : base(n, l, d)
   {
      facultatea = f;
   }

   public override void show()
   { Console.WriteLine("{0} s-a nascut in {1} la data de {2}. Este student(a)
        la Facultatea de {3}", nume, locnas, datanas, facultatea); }
}
class StudentMath : Student
{
   public string specializarea;
   public StudentMath(string n, string l, string d, string f, string s)
      : base(n, l, d, f)
   {
      specializarea = s;
   }
   public override void show()
   { Console.WriteLine("{0} s-a nascut in {1} la data de {2}. Este student(a)
        la Facultatea de {3}, specializarea {4}", nume, locnas, datanas,
        facultatea, specializarea); }
}

class PersoanaDemo
{
   public static void Main()
   {
      Persoana p1 = new Persoana("Vasilescu Vasile", "Iasi", "11 februarie
        1966");
      Student s1 = new Student("Popescu Gina", "Bacau", "22 august
        1984", "Chimie");
      StudentMath sm1 = new StudentMath("Ionescu Ion", "Botosani", "30
        martie 1986", "Matematica", "Matematica-Informatica");
      p1.show();
      Console.WriteLine();
      s1.show();
      Console.WriteLine();
      sm1.show();
   }
}
```

Rezultat:

Vasilescu Vasile s-a nascut in Iasi la data de 11 februarie 1966

Popescu Gina s-a nascut in Bacau la data de 22 august 1984. Este student(a) la Facultatea de Chimie

Ionescu Ion s-a nascut in Botosani la data de 30 martie 1986. Este student(a) la Facultatea de Matematica, specializarea Matematica-Informatica

Utilizarea metodelor abstracte

In unele situatii este necesar sa creati o clasa de baza care stabileste un sablon general pentru toate clasele derivate. O astfel de clasa fixeaza un set de metode, numite metode abstracte, pe care clasele derivate trebuie sa le implementeze.

O metoda abstracta se poate crea specificand modificatorul de tip abstract. Metodele abstracte nu au corp si nu sunt implementate in clasa de baza. Clasele derivate trebuie sa le extinda—nu se admite sa utilizeze versiunea definita de clasa de baza. Metodele abstracte sunt in mod implicit virtuale. In fapt, utilizarea ambilor modificatori virtual si abstract genereaza eroare.

Pentru declararea unei metode abstracte se utilizeaza sintaxa:

```
public abstract tip nume(lista-parametri);
```

Metoda nu are corp, este publica si nu este permisa utilizarea sa in cazul metodelor statice.

O clasa care contine o metoda abstracta trebuie declarata ea insasi abstracta, precedand declaratia class cu specificatorul abstract. Deoarece clasele abstracte nu definesc implementari complete, nu se pot instantia. Incercarea de a crea un obiect cu ajutorul operatorului new, genereaza eroare.

Atunci cand o clasa derivata mosteneste o clasa abstracta trebuie sa implementeze toate metodele abstracte din clasa de baza.

Exemplul 11. Utilizarea metodelor abstracte

```
using System;
abstract class Forma2D
{
   public int lungime;
   public int inaltime;
   public string nume;

   public Forma2D(int l, int i, string n)
   { lungime = l; inaltime = i; nume = n; }

   public abstract double aria();
}
class Triunghi : Forma2D
{
   public string style;
   public Triunghi(int l, int i, string n, string s)
      : base(l, i, n)
   {
      style = s;
   }
   public override double aria()
   {
      return lungime * inaltime / 2;
   }

}
class Dreptunghi : Forma2D
{
   public Dreptunghi(int l, int i, string n)
      : base(i, i, n)
   {  }
   public override double aria()
   {
      return lungime * inaltime;
   }
}
class DemoAbs
{
   public static void Main()
   {
      Triunghi t1 = new Triunghi(8, 4, "triunghi", "dreptunghic
      isoscel");
      Triunghi t2 = new Triunghi(5, 7, "triunghi", "oarecare");
      Dreptunghi d1 = new Dreptunghi(3, 4, "dreptunghi");
      Console.WriteLine("Obiectul t1 este un  {0} {1} avand
      aria={2}", t1.nume, t1.style, t1.aria());
      Console.WriteLine("Obiectul t2 este un  {0} {1} avand
      aria={2}", t2.nume, t2.style, t2.aria());
      Console.WriteLine("Obiectul d1 este un  {0} avand aria={1}",
      d1.nume, d1.aria());
//Forma2D f2d=new Forma2D(); Eroare!, Forma2D este abstracta
   }
}
```

Rezultat:

Obiectul t1 este un triunghi dreptunghic isoscel avand aria=16
Obiectul t2 este un triunghi oarecare avand aria=17
Obiectul d1 este un dreptunghi avand aria=16

Utilizarea cuvantului cheie sealed

In cazul in care doriti ca o clasa sa nu fie mostenita, utilizati cuvantul cheie sealed. Declaratia clasei trebuie precedata de sealed. Spre exemplu: sealed class A { } nu permite mostenirea clasei A. Este incorect sa declarati o clasa abstract si sealed.

Clasa object

Limbajul C# defineste o clasa speciala, numita object, care este clasa de baza implicita pentru toate celelalte clase si pentru toate celelalte tipuri, inclusiv tipurile valorice. Astfel, toate celelalte tipuri deriva din clasa object. Din punct de vedere tehnic, numele C# object este sinonim pentru System.Object, din arhitectura .NET.

Clasa object defineste mai multe metode, disponibile pentru orice obiect. Spre exemplu:

public virtual bool Equals (object ob)	Determina daca obiectul apelant este acelasi ca si cel referit de ob
public static bool Equals(object ob1, object ob2)	Determina daca ob1 si ob2 refera acelasi obiect.
public virtual string ToString()	Intoarce un string care descrie obiectul. Aceasta metoda se apeleaza automat atunci cand obiectul este afisat utilizand WriteLine().
protected Finalize()	Executa actiunile de distrugere a obiectului, anterioare colectarii spatiului ocupat de acesta. Finalize() se acceseaza prin intermediul destructorilor.
protected object MemberwiseClone()	Efectueaza o copie superficiala a obiectului. Acesta contine toti membrii obiectului, dar nu si obiectele pe care acestia le refera.

Exemplele 12 si 13. Metode ale clasei object in actiune

```
using System;
class Persoana
{
   public string nume;     public string datanas;
   public Persoana(string n, string d)
   {       nume = n; datanas = d;    }
}
class Impdesp
{    public static void Main()
   {   string n = "Ana";    string d = "1975";     bool a;
       Persoana p1 = new Persoana(n, d);
       Persoana p2 = new Persoana(n, d);
       Persoana p3;
       p3 = p1;
       a = Equals(p1,p2);
       Console.WriteLine("a={0}",a);

       a=p1.Equals(p2);
       Console.WriteLine("a={0}",a);

       a = Equals(p1, p3);
       Console.WriteLine("a={0}", a);

       int i = 10, j=10;
       a = Equals(i, j);
       Console.WriteLine("a={0}",a);
   }
}
```

Rezultat:

a=False
a=False
a=True
a=True

```
using System;
class MyClass
{
   static int count = 1;
   int id;
   public MyClass()
   {
      id = count;
      count++;
   }
   public override string ToString()
   {
     return "Instanta MyClass #"+id;
   }
}
class Test
{
   public static void Main()
   {
      MyClass ob1 = new MyClass();
      MyClass ob2 = new MyClass();
      MyClass ob3 = new MyClass();
      Console.WriteLine(ob1);
      Console.WriteLine(ob2);
      Console.WriteLine(ob3);
   }
}
```

Rezultat:

Instanta MyClass #1
Instanta MyClass #2
Instanta MyClass #3

Metode de extindere

Functionalitatea unor clase poate fi extinsa fara a include alte campuri sau membrii clasei respective.

Acest lucru este posibil daca este implementata o metoda de extinedere in orice alta clasa.

Se utilizeaza cuvantul cheie this (ca in exemplul alaturat).

Daca metoda primeste si alti parametrii atunci acestia sunt pozitionati dupa this (vezi exemplul).

In exemplul alaturat functionalitatea stringurilor este extinsa de metoda MetodaDeExtindere.

Observatie. Pentru compilare, este necesar referirea asamblajului System.Core.dll

```
using System;
namespace SpatiuDeNume
{
    public static class ExtindereClass
    {
        public static void MetodaDeExtindere( this
       string obiect, int a)
        {
            Console.WriteLine("Aceasta metoda
       extinde clasa string");
            Console.WriteLine(obiect);
            Console.WriteLine(a);
        }
    }
    class Demo
    {
        public static void Main()
        {
            string s = "Un string.";
            s.MetodaDeExtindere(5);
        }
    }
}
```

Impachetare si despachetare

Toate tipurile valorice sunt derivate din object. Asadar o referinta de tip object se poate utiliza pentru a referi orice alt tip, inclusiv un tip valoric. *Atunci cand o referinta de tip object refera un tip valoric, are loc un fenomen numit impachetare*. Impachetarea determina memorarea valorii apartinand unui tip valoric intr-un obiect. Astfel, tipul valoric este impachetat intr-un obiect. Obiectul poate fi folosit ca oricare altul. *Impachetarea se realizeaza prin atribuirea valorii dorite unei referinte de tip obiect.*

Despachetarea este operatia de determinare a valorii continute de un obiect. Aceasta se utilizeaza folosind un cast al referintei obiectului la tipul dorit.

Programele alaturate demonstreaza impachetarea si despachetarea. In cel de-al doilea program se observa cum valoarea 10 este automat impachetata intr-un obiect.

Exemplele 14 si 15. Impachetare si despachetare

```
using System;
class BoxDemo
{
   public static void Main()
   {
      int x;
      object ob;
      x = 10;
      ob = x; //impachetam x intr-un obiect

      int y = (int)ob; //despachetam obiectul ob intr-o
      //valoare de tip int
      Console.WriteLine(y);
   }
}
```

Rezultat:

10

```
using System;
class BoxDemo
{
   public static void Main()
   {
      string l;
      l = 10.ToString();
      Console.WriteLine(l);
   }
}
```

Rezultat:

10

Interfete, Delegari, Evenimente

Definirea si implementarea interfetelor

O interfata *defineste un set de metode care vor fi implementate de una sau mai multe clase*. O *interfata nu implementeaza metode* ci doar precizeaza ce anume va contine o clasa care implementeaza interfata. Din punct de vedere sintactic, interfetele sunt similare claselor abstracte. Insa, exista mai multe aspecte care le diferentiaza. Spre exemplu: daca in clasele abstracte unele metode erau implementate iar altele nu*, in cazul interfetelor toate metodele nu pot avea corp (nu sunt implementate)*.

O interfata poate contine doar: *metode, proprietati, evenimente si indexari*. Interfetele nu pot contine variabile, constructori sau destructori.

Daca o clasa nu poate mosteni decat o clasa, in schimb o *clasa poate implementa oricate interfete*. De asemenea, o interfata poate fi implementata de oricate clase. Asadar, este posibil ca doua clase sa implementeze aceeasi interfata in moduri diferite.

Daca o clasa implementeaza o interfata atunci *aceasta trebuie sa implementeze toti membrii interfetei.* Prin intermediul interfetelor, limbajul C# permite beneficierea la maximum de *aspectul "o singura interfata mai multe metode" al polimorfismului.*

l Interfetele se declara utilizand cuvantul cheie interface. O forma simplificata a unei interfete (contine doar metode) este:

```
acces interface nume
{ tip-rez nume-metoda1 (lista-param);
  tip-rez nume-metoda2 (lista-param);
  //….
  tip-rez nume-metodaN (lista-param); }
```

unde nume reprezinta numele intefetei. De remarcat faptul ca metodele sunt declarate utilizand numai tipul lor si signatura. *Metodele unei interfete sunt in mod implicit publice, nefiind permisa prezenta modificatorilor de acces*. De asemenea, *metodele interfetelor nu pot fi declarate ca statice*.

Dupa ce o interfata a fost declarata, una sau mai multe clase (sau structuri) o pot implementa. Forma generala a unei clase care implementeaza o interfata este:

```
class nume-clasa: nume-interfata
{//corpul clasei}
```

Daca o clasa (sau o structura) implementeaza o interfata atunci ea trebuie sa implementeze toti membrii interfetei. Daca o clasa implementeaza mai multe interfete, atunci numele interfetelor sunt separate prin virgula (class nume-clasa: nume-interfata1, nume-interfata2,…,nume-interfataN).

O clasa poate sa mosteneasca o clasa de baza si sa implementeze mai multe interfete. In acest caz, *numele clasei trebuie sa fie primul in lista separata prin virgule. Metodele care implementeaza o interfata trebuie declarate publice*, deoarece metodele sunt in mod implicit publice in cadrul interfetei. De asemenea, tipul si signatura metodei din cadrul clasei trebuie sa se potriveasca cu tipul si signatura metodei interfetei.

Exemplul 1. Utilizarea unei interfete

```
using System;
public interface Forma2D
{
   double Aria();      double LungFrontiera();
}
public class Cerc: Forma2D
{
   public double raza;     private const float PI = 3.14159f;
   public double Aria()    { return (PI * raza * raza);}
   public double LungFrontiera()    {return (2*PI*raza);}
   public Cerc(double r)    {raza=r;}
 }
public class Patrat : Forma2D
{
   public double latura;
   public double Aria()    {return (latura * latura);}
   public double LungFrontiera()    { return (4 * latura);}
   public Patrat(double l)    {latura = l;}
}
class IterfDemo
{   public static void Main()
   {   Cerc c = new Cerc(3);       Patrat p = new Patrat(3);
     Console.WriteLine("Afiseaza informatii despre cerc:\naria={0:#.##} \t lungimea frontierei={1:#.##}", c.Aria(),
      c.LungFrontiera());
       Console.WriteLine("\nAfiseaza informatii despre patrat:\naria={0:#.##} \t lungimea frontierei={1:#.##}",
      p.Aria(), p.LungFrontiera());
   }
}
```

Rezultat:

Afiseaza informatii despre cerc:
aria=28,27   lungimea frontierei=18,85
Afiseaza informatii despre patrat:
aria=9   lungimea frontierei=12

Referinte avand ca tip o interfata

Desi pare surprinzator, *puteti declara o variabila referinta avand ca tip o interfata*. O astfel de variabila poate referi orice obiect care implementeaza interfata.

La apelul unei metode prin intermediul referintei catre interfata, se va executa versiunea metodei care este implementata de obiectul apelant.

Exemplul 2

```
using System;
public interface Forma2D
{
   double Aria();
   double LungFrontiera();
}
public class Cerc : Forma2D
{
   public double raza; private const float PI = 3.14159f;

   public double Aria() { return (PI * raza * raza); }

   public double LungFrontiera() { return (2 * PI * raza); }

   public Cerc(double r) { raza = r; }
}
public class Patrat : Forma2D
{
   public double latura;

   public double Aria() { return (latura * latura); }

   public double LungFrontiera() { return (4 * latura); }

   public Patrat(double l) { latura = l; }
}
class IterfDemo
{
   public static void Main()
   {
      Cerc c = new Cerc(3);    Patrat p = new Patrat(3);

      Console.WriteLine("Afiseaza informatii despre cerc:");
      DisplayInfo(c);

      Console.WriteLine("\nAfiseaza informatii despre patrat:");
      DisplayInfo(p);
   }
   static void DisplayInfo(Forma2D f)
   {Console.WriteLine("aria={0:#.##} \t lungimea
      frontierei={1:#.##}", f.Aria(), f.LungFrontiera());}
}
```

Rezultat:

Afiseaza informatii despre cerc:
aria=28,27   lungimea frontierei=18,85
Afiseaza informatii despre patrat:
aria=9   lungimea frontierei=12

Utilizarea proprietatilor in interfete

Ca si metodele, *proprietatile se pot specifica in cadrul unei interfete fara a include corpul.*

In cazul proprietatilor accesibile atat la scriere cat si la citire vor aparea ambii accesori (get si set), in tip ce pentru proprietatile accesibile doar la citire (scriere) va aparea numai accesorul get (set).

Exemplul 3. Utilizarea unei proprietati accesibila la citire in cadrul unei interfete

```
using System;
public interface Forma2D
{
    double Aria();
    double LungFrontiera();
    string denumire { get; }
}
public class Cerc : Forma2D
{
    string s = "cerc";
    public double raza; private const float PI = 3.14159f;

    public double Aria() { return (PI * raza * raza); }
    public double LungFrontiera() { return (2 * PI * raza); }

    public string denumire{ get{return s;}}

    public Cerc(double r) { raza = r; }
}
public class Patrat : Forma2D
{
    string s="patrat";
    public double latura;

    public double Aria() { return (latura * latura); }
    public double LungFrontiera() { return (4 * latura); }
    public string denumire { get {return s; } }

    public Patrat(double l) { latura = l; }
}
class IterfDemo
{
    public static void Main()
    {
        Cerc c = new Cerc(3); Patrat p = new Patrat(3);
        Console.WriteLine("Afiseaza informatii despre {0}:", 
        c.denumire);
        DisplayInfo(c);
        Console.WriteLine("\nAfiseaza informatii despre 
        {0}:", p.denumire);
        DisplayInfo(p);
    }
    static void DisplayInfo(Forma2D f)
    {Console.WriteLine("aria={0:#.##} \t lungimea 
        frontierei={1:#.##}", f.Aria(), f.LungFrontiera());}
}
```

Rezultat:

```
Afiseaza informatii despre cerc:
aria=28,27          lungimea frontierei=18,85
Afiseaza informatii despre patrat:
aria=9              lungimea frontierei=12
```

Implementari explicite

In exemplele anterioare totul a decurs firesc in ceea ce priveste implementarea interfetelor. Pot aparea insa o serie de probleme atunci cand se doreste implementarea mai *multor interfete care contin metode cu aceeasi denumire.*

Spre exemplu, daca o clasa implementeaza doua interfete care contin o metoda avand acelasi nume (sa zicem Display()) atunci o singura implementare a metodei Display() satisface ambele interfete.

Exista totusi situati cand doriti sa implementati in mod independent metoda Display() pentru ambele interfete. In acest caz, trebuie sa implementati interfetele in mod explicit*. O implementare explicita este realizata prin includerea numelui interfetei impreuna cu numele metodei*.

La realizarea unei implementari explicite *nu trebuie apelat modificatorul de acces public*. De fapt*, prin implementarea explicita, metoda devine practic private si nu poate fi accesata din afara clasei decat de o referinta de tipul interfetei sau utilizand un cast* (a se vedea exemplul 4). Practic, am implementat o metoda si in acelasi timp am ascuns-o.

Exemplul 4 prezinta aceste aspecte.

Exemplul 4. Implementari explicite

```
using System;
public interface Forma2D
{    double Aria(); double LungFrontiera();
   void Display();  string denumire { get; }  }

public interface Forma2DDisplay
{    void Display(); }

public class Cerc : Forma2D, Forma2DDisplay
{
   string s = "cerc";
   public double raza; private const float PI = 3.14159f;

   public double Aria() { return (PI * raza * raza); }
   public double LungFrontiera() { return (2 * PI * raza); }

   void Forma2D.Display()
   {
      Console.WriteLine("Afiseaza informatii despre {0}:",
      denumire);
      Console.WriteLine("aria={0:#.##}", Aria());
      Console.WriteLine("lungimea frontierei={0:#.##}",
      LungFrontiera());
   }
   void Forma2DDisplay.Display()
   {
      Console.WriteLine("Aceasta metoda ar putea furniza
      informatii despre cerc");
   }
   public string denumire { get { return s; } }

   public Cerc(double r) { raza = r; }
}
```

```
public class Patrat : Forma2D, Forma2DDisplay
{    string s="patrat";    public double latura;
   public double Aria() { return (latura * latura); }
   public double LungFrontiera() { return (4 * latura); }
   void Forma2D.Display()
   {    Console.WriteLine("Afiseaza informatii despre {0}:",
        denumire);
        Console.WriteLine("aria={0:#.##}", Aria());
        Console.WriteLine("lungimea frontierei={0:#.##}",
        LungFrontiera());    }
   void Forma2DDisplay.Display()
   {     Console.WriteLine("Aceasta metoda ar putea
        furniza informatii despre patrat");    }
   public string denumire { get {return s; } }
   public Patrat(double l) { latura = l; }
}
class IterfDemo
{    public static void Main()
   {  Cerc c = new Cerc(3); Patrat p = new Patrat(3);
      Forma2D f1 = (Forma2D)c;        f1.Display();
      Forma2DDisplay f2 = (Forma2DDisplay)c;
      f2.Display();
      Console.WriteLine();
      Forma2D f3 = (Forma2D)p;        f3.Display();
      Forma2DDisplay f4 = (Forma2DDisplay)p;
      f4.Display();
 } }
```

Rezultat:

```
Afiseaza informatii despre cerc:
aria=28,27
lungimea frontierei=18,85
Aceasta metoda ar putea furniza informatii despre cerc

Afiseaza informatii despre patrat:
aria=9
lungimea frontierei=12
Aceasta metoda ar putea furniza informatii despre patrat
```

Interfetele pot fi mostenite

O interfata poate fi mostenita de o alta interfata. Sintaxa este comuna cu cea care se utilizeaza la mostenirea claselor.

Daca o clasa implementeaza o interfata care mosteneste o alta interfata, clasa trebuie sa contina implementari pentru toti membrii definiti pe lantul de mostenire.

Exemplul 5 ilustreaza aceste aspecte.

Daca incercati sa eliminati implemetarea metodei Metoda1() din clasa Myclass, se va produce o eroare la compilare.

Exemplul 5. Implementarea unui lant de mostenire

```
using System;
public interface A
{    void Metoda1();  }
public interface B : A
{    void Metoda2();    void Metoda3(); }
class Myclass : B
{
    public void Metoda1()
    { Console.WriteLine("Implementarea metodei 1"); }
    public void Metoda2()
    { Console.WriteLine("Implementarea metodei 2");  }
    public void Metoda3()
    { Console.WriteLine("Implementarea metodei 3"); }
}
class MostenireDemo
{
    public static void Main()
    {
        Myclass ob = new Myclass();
        ob.Metoda1();
        ob.Metoda2();
        ob.Metoda3();
    }
}
```

Rezultat:

Implementarea metodei 1
Implementarea metodei 2
Implementarea metodei 3

Delegari

O delegare reprezinta un tip referinta care *executa metode avand acelasi format (acelasi tip rezultat si acelasi numar si tip de parametrii)*.

Intr-un limbaj mai detaliat, chiar daca o metoda nu este un obiect, ea ocupa o locatie in memoria fizica. La invocarea metodei, controlul se transfera la adresa punctului de intrare in metoda. Aceasta adresa poate fi atribuita unei delegari si astfel metoda poate fi apelata prin intermediul delegarii. Mai mult, acceasi delegare poate fi utilizata pentru a apela si alte metode, modificand pur si simplu metoda referita.

Delegarile sunt similare pointerilor catre functii in C si C++.

Delegarile se declara utilizand cuvantul cheie delegate. Forma generala a unei delegari este:

acces delegate tip-rez nume (lista-parametrii);

unde tip-rez este tipul valorii intoarse de metodele pe care delegarea le va apela, numele delegarii este specificat prin nume, iar parametrii necesari metodelor care vor fi apelate prin intermediul delegarii sunt specificati prin lista-parametrii.

Delegarile sunt importante din *doua motive.* In primul rand, *delegarile permit implementarea evenimentelor*, iar in al doilea rand, *delegarile amana determinarea metodei invocate pana la momentul executiei*. Aceasta din urma capacitate se dovedeste utila atunci cand creati o arhitectura care permite adaugarea componentelor pe parcurs.

Exemplele 6 si 7. Delegarile pot apela atat metode statice (exemplul 6) cat si metode ale obiectelor (exemplul 7)

```
using System;
public delegate double Mydelegate(double a);
class DelegateDemo
{
    const float PI = 3.14159f;

    static double Aria(double r)
    { return (PI * r * r); }

    static double LungFrontiera(double r)
    { return (2 * PI * r); }

    public static void Main()
    {
        double raza=3;

        Mydelegate del = new Mydelegate(Aria);

        Console.WriteLine("Aria= {0:#.##}", del(raza));

        del = new Mydelegate(LungFrontiera);

        Console.WriteLine("Lungimea frontierei={0:#.##}",
        del(raza));
    }
}
```

Rezultat:
Aria=28,27
Lungimea frontierei=18,85

```
using System;
public delegate double Mydelegate();
public class Cerc
{
    public double raza; private const float PI = 3.14159f;

    public double Aria()
    { return (PI * raza * raza); }

    public double LungFrontiera()
    { return (2 * PI * raza); }

    public Cerc(double r) { raza = r; }
}

class DelegateDemo
{
    public static void Main()
    {
        Cerc c = new Cerc(3);
        Mydelegate del = new Mydelegate(c.Aria);
        Console.WriteLine("Aria= {0:#.##}", del());
        del = new Mydelegate(c.LungFrontiera);
        Console.WriteLine("Lungimea frontierei={0:#.##}",
        del());
    }
}
```

Rezultat:
Aria=28,27
Lungimea frontierei=18,85

Multicasting

Una dintre cele mai interesante facilitati oferite de delegari o reprezinta *capacitatea de multicasting. Multicastingul reprezinta capacitatea de a crea un lant de metode care vor fi automat apelate la invocarea unei delegari.*

Un astfel de lant este usor de creat. Se instantiaza mai intai o delegare, iar apoi utilizand operatorul += se adauga metode in lant sau utilizand operatorul -= se elimina metode din lant.

Exista o restrictie importanta, si anume: *delegarile multicast trebuie sa intoarca un rezultat de tip void*.

Exemplul alaturat rescrie exemplul 6 modificand tipul intors si utilizand parametrul out pentru a intoarce aria si lungimea frontierei in modulul apelant.

Exemplul 8. Multicasting

```
using System;
public delegate void Mydelegate(double r, out double a);
class DelegateDemo
{   const float PI = 3.14159f;
    static void Aria(double r,out double a)
    { a=PI * r * r; }
    static void LungFrontiera(double r, out double lf)
    { lf=2 * PI * r; }
    static void Cerc(double r, out double b)
    { Console.WriteLine("Cercul de raza r={0} are", r);
    b = 0;  }

    public static void Main()
    {
        double raza=3, a,lf;
        Mydelegate del = new Mydelegate(Cerc);
        del += new Mydelegate(Aria);
        del(raza, out  a);
        Console.WriteLine("Aria= {0:#.##}", a);

        del += new Mydelegate(LungFrontiera);
        del(raza, out lf);
        Console.WriteLine("Lungimea frontierei={0:#.##}",
        lf);
    }
}
```

Rezultat:

Cercul de raza r=3 are
Aria= 28,27
Cercul de raza r=3 are
Lungimea frontierei=18,85

Evenimentele

Pe fundamentul reprezentat de delegari, limbajul C# a construit o alta facilitate importanta: evenimentele.

Un *eveniment* reprezinta, in esenta, *notificarea automata din partea unei clase ca s-a produs o actiune in program*. Pe baza acestei instiintari puteti raspunde cu o rutina pentru tratarea acestei actiuni.

Evenimentele functioneaza dupa urmatorul mecanism: *un obiect care este intresat de un eveniment isi inregistreaza o rutina de tratare a acelui eveniment. Atunci cand evenimentul se produce, se apeleaza toate rutinele inregistrate*. Rutinele de tratare a evenimentelor sunt reprezentate prin delegari.

Cele mai frecvente exemple de procesare a evenimentelor sunt intalnite atunci o fereastra de dialog este afisata pe ecran si utilizatorul poate realiza diverse actiuni: *click pe un buton, selectare a unui meniu, tastare a unui text* etc. Atunci cand utilizatorul realizeaza una dintre aceste actiuni, se produce un eveniment. Rutinele de tratare a evenimentelor reactioneaza in functie de evenimentul produs.

Evenimentele sunt entitati membre ale unei clase si se declara utilizand cuvantul cheie event. Forma generala a declaratiei este:

public event delegare-eveniment nume-eveniment;

unde delegare-eveniment reprezinta *numele delegarii utilizate pentru tratarea evenimentului*, iar nume-eveniment *este numele instantei eveniment create*.

Pentru crearea unui eveniment sunt necesare parcurgerea urmatoarelor etape:

(a) setarea delegarii corespunzatoare evenimentului (*the delegate*);

(b) crearea unei clase pentru a transmite argumente (parametrii) rutinei de tratare a evenimentului;

(c) declararea codului corespunzator evenimentului (*the event*);

(d) crearea rutinei de tratare a evenimentului (codul care este executat ca raspuns pentru eveniment, *the handler);*

(e) lansarea evenimentului (*firing the event*).

In exemplul urmator sunt prezentate aceste etape, cu exceptia etapei (b) care nu este necesata deoarece rutina care trateaza evenimentul nu are parametrii. Observati ca pe langa aceste etape, sunt necesare si o serie de alte actiuni care sunt explicate in textul programului.

Exemplul 9. Evenimente

```
using System;
public delegate void MyEventHandlerDelegate(); //(a)
class MyEvent
{
    public event MyEventHandlerDelegate activare; //(c)

    public void Fire()           //Crearea unei metode care genereaza evenimentul
    {
            activare();
    }
}
class EventDemo
{
    static void HandlerEv()        //(d)
    {
        Console.WriteLine("Evenimentul s-a produs");
    }
    public static void Main()
    {
        MyEvent ev = new MyEvent();  //Crearea instantei eveniment
        ev.activare += new MyEventHandlerDelegate(HandlerEv);  //Adaugarea rutinei de tratare in lant

        ev.Fire();                 //(e)
    }
}
```

Rezultat:

Evenimentul s-a produs

Exemplul 10. Evenimentele pot fi multicast. Aceasta permite ca mai multe obiecte sa poata raspunde la instiintarea aparitiei unui eveniment. In exemplul de mai jos este prezentat un exemplu de eveniment multicast.

```
using System;
public delegate void MyEventHandlerDelegate();
class MyEvent
{   public event MyEventHandlerDelegate activare;
    public void Fire()
    {       if (activare != null)  activare();    } }
class A {   public void Ahandler()
    {Console.WriteLine("Eveniment tratat si de metoda Ahandler");}}
class B {   public void Bhandler()
    {Console.WriteLine("Eveniment tratat si de metoda Bhandler");} }
class EventDemo {
    static void HandlerEv()
    {Console.WriteLine("Evenimentul s-a produs");}
    public static void Main()
    {   MyEvent ev = new MyEvent();
        A obA = new A();     B obB = new B();
        ev.activare += new MyEventHandlerDelegate(HandlerEv);
        ev.activare += new MyEventHandlerDelegate(obA.Ahandler); //adaugarea metodei Ahandler la lant
        ev.activare += new MyEventHandlerDelegate(obB.Bhandler); //adaugarea metodei Bhandler la lant
        ev.Fire();
        Console.WriteLine();
              ev.activare -= new MyEventHandlerDelegate(obA.Ahandler); //eliminarea metodei Bhandler din lant
        ev.Fire();     }
}
```

Rezultat:

```
Evenimentul s-a produs
Eveniment tratat si de metoda Ahandler
Eveniment tratat si de metoda Bhandler

Evenimentul s-a produs
Eveniment tratat si de metoda Bhandler
```

Observatii:

Evenimentele se utilizeaza, de regula, la crearea aplicatiilor Windows. Pentru a usura munca utilizatorului, C# permite utilizarea unor clase care pun la dispozitie evenimente si delegari standard. Astfel:

1) spatiul de nume System contine o delegare standard, intitulata EventHandler [mai precis forma sa generala este public delegate void EventHandler(object sender, EventsArgs e)]. Aceasta primeste doi parametrii. Primul parametru, object sender, contine sursa (obiectul) care genereaza evenimentul iar cel de-al doilea argument este un obiect dintr-o clasa standard, intitulata EventArgs din spatiul de nume System. Aceasta clasa este utilizata pentru a transmite argumente (parametrii) rutinei de tratare a evenimentului (clasa amintita la punctul (b) din al doilea slide referitor la evenimente);

2) intrucat EventHandler are doi parametrii, urmeaza ca metodele care trateaza evenimentul contin aceiasi parametri ca si delegarea;

3) spatiul de nume System.Windows.Forms pune la dispozitie un numar mare de evenimente standard. Acestea sunt asociate diverselor controlere (button, label, radio button, etc.). In Visual C#, lista evenimentelor se gaseste in meniul Properties.

Exemplul urmator ilustreaza utilizarea evenimentelor si rutinelor de tratare ale acestora in cazul unei aplicatii windows.

```
using System;
using System.Windows.Forms;
using System.Drawing;

public class MyForm : Form
{
private Label myDateLabel;
private Button btnUpdate;

public MyForm()
{ InitializeComponent();}

//INSEREAZA METODA InitializeComponent() AICI

protected void btnUpdate_Click( object sender, System.EventArgs e)
{ DateTime currentDate =DateTime.Now ;
this.myDateLabel.Text = currentDate.ToString(); }

protected void btnUpdate_MouseEnter(object sender, System.EventArgs e)
{ this.BackColor = Color.HotPink;}

protected void btnUpdate_MouseLeave(object sender, System.EventArgs e)
{ this.BackColor = Color.Blue;}

protected void myDataLabel_MouseEnter(object sender, System.EventArgs e)
{ this.BackColor = Color.Yellow;}

protected void myDataLabel_MouseLeave(object sender, System.EventArgs e)
{ this.BackColor = Color.Green; }

public static void Main( string[] args )
{ Application.Run( new MyForm() );   /* creaza fereastra*/      }
}
```

```
private void InitializeComponent()
 { this.Text = Environment.CommandLine;
 this.StartPosition = FormStartPosition.CenterScreen;
 this.FormBorderStyle = FormBorderStyle.Fixed3D;

 myDateLabel = new Label(); // Creaza label

 DateTime currentDate = new DateTime();
 currentDate = DateTime.Now;
 myDateLabel.Text = currentDate.ToString();

 myDateLabel.AutoSize = true;
 myDateLabel.Location = new Point( 50, 20);
 myDateLabel.BackColor = this.BackColor;

 this.Controls.Add(myDateLabel); // Adauga label-ul ferestrei

 this.Width = (myDateLabel.PreferredWidth + 100); // Seteaza latimea ferestrei pe baza latimii labelui

 btnUpdate = new Button(); // Creaza button

 btnUpdate.Text = "Update";
 btnUpdate.BackColor = Color.LightGray;
 btnUpdate.Location = new Point(((this.Width/2) -
(btnUpdate.Width / 2)), (this.Height - 75));

 this.Controls.Add(btnUpdate); // Adauga button-ul ferestrei

 btnUpdate.Click += new System.EventHandler(this.btnUpdate_Click);
 btnUpdate.MouseEnter += new System.EventHandler(this.btnUpdate_MouseEnter);
 btnUpdate.MouseLeave += new System.EventHandler(this.btnUpdate_MouseLeave);
 myDateLabel.MouseEnter += new System.EventHandler(this.myDataLabel_MouseEnter);
 myDateLabel.MouseLeave += new System.EventHandler(this.myDataLabel_MouseLeave);
 }
```

Spatii de nume

Spatii de nume

Un spatiu de nume defineste un domeniu de valabilitate, separand astfel un set de clase, variabile, metode, etc. de un alt set.

Spatiul de nume pe care l-am utilizat frecvent pana in momentul de fata este System. Din acest motiv, progamele utilizate pana in prezent au inclus linia using System; Arhitectura .NET pune la dispozitie o serie larga de spatii de nume (spre exemplu System.IO, System.Windows.Forms; etc.)

Spatiile de nume sunt importante deoarece in aplicatii sunt utilizate o gama mare de variabile, metode, proprietati, clase avand diverse nume. Acestea se refera la functiile de biblioteca, la codul dezvoltat de terti sau la codul dezvoltat de utilizator. Fara spatiile de nume, aceste nume de variabile, metode, etc. ar putea intra in conflict daca sunt aceleasi.

Spatiile de nume se declara utilizand cuvantul cheie namespace. Forma generala a declaratiei namespace este:

```
namespace nume
{//membrii}
```

unde nume reprezinta numele spatiului.

In cadrul unui spatiu de nume se pot declara *clase, structuri, delegari, enumerari, interfete sau chiar un alt spatiu de nume.*

Exemplul alaturat demonstreaza utilizarea spatiului de nume. Deoarece clasa Point este declarata in cadrul spatiului de nume Puncte, pentru crearea unei instante a clasei Point trebuie utilizat spatiul de nume si operatorul punct (.) Dupa ce s-a creat obiectul de tipul Point, nu mai este necesar sa calificam obiectul sau oricare din membrii sai cu numele spatiului.

Exemplul 12. Spatii de nume

```
using System;
namespace Puncte
{
class Point
{
    public double x;
    public double y;
    public Point(double xx, double yy)
    {
        x = xx; y = yy;
    }

}
}
class Segmdr
{
    public static void Main()
    {
        Puncte.Point punct1 = new Puncte.Point(3,4);
        Puncte.Point punct2 = new Puncte.Point(5,6);
        double dist;

        dist = Math.Sqrt((punct1.x - punct2.x) * (punct1.x -
        punct2.x) + (punct1.y - punct2.y) * (punct1.y -
        punct2.y));
        Console.WriteLine("Distanta dintre punctele ({0},{1})
        si ({2},{3}) este: {4:#.##}", punct1.x, punct1.y,
        punct2.x, punct2.y, dist);
    }
}
```

Rezultat:

Distanta intre punctele (3,4) si (5,6) este: 2.83

Daca programul utilizatorului include referinte frecvente catre membrii unui spatiu de nume, specificarea spatiului de nume ori de cate ori trebuie sa referiti un membru al sau devine greoaie. Directiva using rezolva aceasta problema. Directiva using are doua forme: using nume; si respectiv using alias=nume; Referitor la cea de-a doua forma, alias devine un alt nume pentru clasa sau spatiul de nume specificat prin nume. Ambele forme ale directivei using sunt specificate in exemplele de mai jos, pentru care rezultatele sunt acelasi ca in exemplul 12.

Exemplul 13

```
using System;
using Puncte;
namespace Puncte
{
    class Point
    {
        public double x;
        public double y;
        public Point(double xx, double yy)
        { x = xx; y = yy; }
    }
}
class Segmdr
{
    public static void Main()
    {
        Point punct1 = new Point(3, 4); Point punct2 = new
        Point(5, 6);
        double dist;

        dist = Math.Sqrt((punct1.x - punct2.x) * (punct1.x -
        punct2.x) + (punct1.y - punct2.y) * (punct1.y -
        punct2.y));
        Console.WriteLine("Distanta dintre punctele ({0},{1})
        si ({2},{3}) este: {4:#.##}", punct1.x, punct1.y,
        punct2.x, punct2.y, dist);
    }
}
```

Exemplul 14

```
using System;
using p=Puncte.Point;
namespace Puncte
{
    class Point
    {
        public double x;
        public double y;
        public Point(double xx, double yy)
        { x = xx; y = yy; }
    }
}
class Segmdr
{
    public static void Main()
    {
        p punct1 = new p(3, 4); p punct2 = new p(5, 6);
        double dist;

        dist = Math.Sqrt((punct1.x - punct2.x) * (punct1.x -
        punct2.x) + (punct1.y - punct2.y) * (punct1.y -
        punct2.y));
        Console.WriteLine("Distanta dintre punctele ({0},{1})
        si ({2},{3}) este: {4:#.##}", punct1.x, punct1.y,
        punct2.x, punct2.y, dist);
    }
}
```

Spatiile de nume sunt aditive. Astfel pot exista declaratii namespace cu acelasi nume. Aceasta permite distribuirea unui spatiu de nume in mai multe fisiere sau chiar separarea sa in cadrul aceluiasi fisier. In exemplul de mai jos sunt definite doua spatii de nume cu aceeasi denumire. La compilare, continutul ambelor spatii de nume este adaugat laolalta.

Exemplul 15

```
using System;
namespace Puncte
{   class Point
   {      public double x;        public double y;    }
}
namespace Puncte
{
   class Line
   {
      public Point punct1 = new Point();        public Point punct2 = new Point();
      public double Lung()
      {   double l;
         l = Math.Sqrt((punct1.x - punct2.x) * (punct1.x - punct2.x) + (punct1.y - punct2.y) * (punct1.y - punct2.y));
         return l;
      }
   }
}
class Segmdr
{
   public static void Main()
   {
      Puncte.Line seg = new Puncte.Line();
      double dist;
      seg.punct1.x = 3;       seg.punct1.y = 4;
      seg.punct2.x = 5;       seg.punct2.y = 3;
      dist = seg.Lung();
      Console.WriteLine("Distanta dintre punctele ({0},{1}) si ({2},{3}) este: {4:#.##}", seg.punct1.x, seg.punct1.y,
      seg.punct2.x, seg.punct2.y, dist);
   }
}
```

Depanarea si tratarea erorilor

In programare sunt posibile doua tipuri de erori: *erori fatale* si *erori semantice* (sau *logice*). Erorile fatale includ erori simple care impiedica compilarea aplicatiilor (erori de sintaxa) sau probleme mai serioase care apar la executia programelor (exceptii). Erorile semantice sunt mult mai subtile. Spre exemplu, aplicatia creata esueaza in a inregistra un set de date intr-o baza de date deoarece un anumit camp lipseste, sau inregistreaza setul de date in mod eronat. Erorile de acest tip demonstreaza ca logica programului este defectuasa. Ele se mai numesc si erori logice.

Daca erorile fatale sunt relativ usor de detectat si tratat, de multe ori nici nu stim de existenta vreo unei erori logice pana cand vreun utilizator al aplicatiei noastre reclama ca ceva nu functioneaza cum ar trebui. Avem astfel sarcina de a urmari codul programului si de a afla ce nu functioneaza in mod corect. In astfel de situatii, tehnicile de depanare puse la dispozitie atat de limbajul C# cat si de Visual Studio (VS) sunt de mare ajutor.

Procesul prin care sunt identificate si corectate acele portiuni de cod care nu functioneaza asa cum s-a intentionat initial se numeste *depanare (debugging).*

Asa cum ati observat, puteti executa o aplicatie in doua moduri: cu depanarea activata (Start Debugging sau F5) si cu depanarea dezactivata (Start Without Debugging sau Ctrl+F5).

De asemenea, VS permite construirea *aplicatiilor in doua configuratii: Debug si Release*.

Atunci cand este construita o aplicatie in configuratia Debug, nu este executat doar codul programului ci se intampla si altceva. Blocurile de depanare (debug builds) mentin o informare simbolica privind programul nostru astfel ca VS stie ce se intampla cu fiecare linie de cod executata. Aceaste informatii sunt continute in fisierele .pdb din directoarele Debug. Acest aspect ne permite realizarea urmatoarelor operatii: *transmiterea de informatii catre VS, urmarirea si editarea valorilor variabilelor in timpul executiei aplicatiei, oprirea executiei la anumite puncte marcate in codul programului, executia programului linie cu linie, monitorizarea unei variabile in timpul executiei*, etc.

In configuratia Release, codul aplicatiei este optimizat, iar operatiile enumerate mai sus nu sunt posibile. Blocurile release (Release builds) se executa mai rapid, astfel cand realizati o aplicatie, de regula veti furniza utilizatorilor blocuri release intrucat informatia simbolica pe care blocurile debug le contin nu este necesara.

In cele ce urmeaza vom expune cateva tehnici de depanare. Acestea sunt grupate in doua categorii, dupa modul in care acestea sunt utilizate. Asfel, depanarea se poate realiza fie prin intreruperea executiei programului (modul break) in diverse puncte ale acestuia, fie prin consemnarea unor observatii care sa fie analizate ulterior.

Depanare in modul normal (nonbreak)

Sa consideram urmatoarea secventa de cod:

```
Console.WriteLine(“Functia F() urmeaza a fi apelata”);
F(); //Functia F() realizeaza o serie de operatii
Console.WriteLine(“Functia F() a fost apelata”);
```

Codul de mai sus arata cum putem obtine mai multe informatii privind functia F(). Acest aspect este important in multe situatii, insa conduce la o dezordine in ceea ce priveste informatiile afisate in consola. Mai mult, in cazul aplicatiilor Windows nici nu utilizam consola. O cale alternativa este de a afisa astfel de informatii intr-o fereastra separata, fereastra *Output* care se poate selecta din meniul View. Aceasta fereastra ofera informatii legate de compilarea si executia codului, erorile intalnite in timpul compilarii sau poate fi utilizata pentru a afisa informatii de genul celor prezentate mai sus, utile in depanare.

O alta cale este de a crea un fisier separat (*logging file*) care sa adauge diverse informatii legate de depanare atunci cand aplicatia este executata.

Pentru a scrie text in fereastra *Output* avem doua posibilitati. Fie utilizam metodele puse la dispozitie de clasele Debug sau Trace care fac parte din spatiul de nume System.Diagnostics, fie utilizam o facilitate oferita de Visual Studio, si nu de limbajul C#, si anume anumite “marcaje” in program numite *tracepoints*.

Sa consideram mai intai prima posibilitate. La fel ca in cazul clasei Console, cele doua clase amintite mai sus, pun la dispozitie metodele WriteLine() si Write(), insa si metode precum WriteLineIf(), WriteIf() care contin in plus un parametru de tipul bool. Acestea doua din urma scriu text doar daca *parametrul bolean este true.*

Metodele din clasele Debug si Trace opereaza in acelasi mod, insa cu o diferenta importanta. Metodele din *clasa Debug functioneaza doar cu blocurile debug*, in timp ce metodele din *clasa Trace functioneaza cu oricare din blocuri (release sau debug)*. Astfel, metoda Debug.WriteLine() nu se va compila intr-o versiune release a aplicatiei, in timp ce metoda Trace.WriteLine() se compileaza atat in versiunea release cat si versiunea debug.

Metodele Debug.WriteLine() , Trace.WriteLine() nu functioneaza la fel ca metoda Console.WriteLine(). Acestea opereaza doar asupra unui parametru de tip string, si nu permit sintaxe de genul {0} sau {0:#.##}. Optional, se poate utiliza un al doilea parametru de tip string care afiseaza o categorie pentru textul ce urmeaza a fi afisat in fereastra Output. Forma generala a mesajului afisat este:

<category>: <message>

Spre exemplu, urmatoarea linie de cod:

Debug.WriteLine("Aduna i cu j","FunctiaSuma")

afiseaza:

FunctiaSuma: Aduna i cu j

Pentru a beneficia de optiunile de afisare de care de bucura metoda Console.WriteLine(), se poate utiliza metoda Format() a clasei string (vezi exemplul urmator).

Construiti urmatoarea aplicatie atat in configuratia debug cat si in configuratia release. Pentru ca aceasta sa poata fi compilata, includeti o referinta catre asamblajul System.

```
using System;    using System.Diagnostics;
using System.Collections;
namespace ConsoleApplication5
{ class Program
  {
   static int ValoareMaxima(int[] intArray, out ArrayList listaIndecsi)
     {     Debug.WriteLine("Cautarea valorii maxime a inceput");
           int valMax = intArray[0];
           Debug.WriteLine(string.Format("Valoarea maxima a fost
initializata cu {0} la elementul cu indexul 0",valMax));
           for (int i = 1; i < intArray.Length; i++)
           {  Debug.WriteLine(string.Format("Acum testam elementul
cu indexul {0} ", i));
               if (intArray[i] > valMax)
               {  valMax = intArray[i];
                  Debug.WriteLine(string.Format("Un nou maxim gasit.
Noua valoare este {0} la elementul cu indexul {1}",valMax,i));
               }
           }
           Trace.WriteLine(string.Format("Valoarea maxima din sir
este: {0} ",valMax));

          Debug.WriteLine(string.Format("Acum testam care sunt
            elementele din sir care au valoarea maxima"));
           listaIndecsi = new ArrayList();
         for (int i = 0; i < intArray.Length; i++)
           {  Debug.WriteLine(string.Format("Acum testam elementul
            cu indexul {0} ", i));
                if (intArray[i] == valMax)
                  {  valMax = intArray[i];
                     listaIndecsi.Add(i);
                     Debug.WriteLine(string.Format("Un nou element
                gasit care are valoarea {0}. Elementul cu indexul {1}",
                valMax,  i));
                  }
              }
             Trace.WriteLine(string.Format("Elementele din sir care au
              valoarea maxima au indecsii:"));
             foreach(int index in listaIndecsi)
             {      Trace.Write(index.ToString()+"\t");        }
             Trace.WriteLine("");
             return valMax;
          }
          static void Main(string[] args)
          {  int[] myArray = { 1, 8, 3, 9, 6, 2, 5, 9, 3, 9, 0, 2 };
             ArrayList valMaxIndecsi=new ArrayList();
             int maxVal = ValoareMaxima(myArray, out
              valMaxIndecsi);
             Console.WriteLine("Valoarea maxima in sir este: {0}",
              maxVal);
             Console.WriteLine("Elementele din sir care au valoarea
              maxima au indecsii:");
             foreach (int index in valMaxIndecsi)
             {  Console.Write(index.ToString() + "\t");   }
             Console.WriteLine("");
             Console.ReadKey();
          }
      }
  }
```

Tracepoints. Asa cum s-a mentionat mai sus, putem scrie informatii in fereastra Output utilizand o serie de marcaje (tracepoints), facilitate pusa la dispozitie de VS.

Pentru a insera un marcaj tracepoint in codul programului, pozitionati cursorul pe linia dorita (marcajul facut va fi procesat inainte ca linia de cod sa fie executata), click dreapta si selectati Breakpoint ->Insert Tracepoint. In fereastra de dialog care apare pe ecran inserati stringul dorit. Daca doriti sa afisati valoarea unei variabile atunci includeti numele variabilei in acolade. De asemenea, puteti obtine si alte informatii importante utilizand cuvinte cheie precum $FUNCTION, $CALLER, etc. sau este posibil sa executati un macro. O alta facilitate este aceea de a intrerupe executia programului (deci sa actioneze ca un breakpoint). Prin apasarea butonului OK, un romb rosu va aparea in stanga liniei de cod.

Executand un program care are inserate marcaje tracepoints, vom obtine aceleasi rezultat ca atunci cand utilizam clasele Debug si Trace in modul Debug (vezi exemplul precedent). Puteti sterge un astfel de marcaj sau sa-l dezactivati prin click dreapta si Disable Breakpoint.

Principalul avantaj pe care il ofera utilizarea marcajelor este rapiditatea si usurinta cu care sunt adaugate aplicatiei. Drept bonus este obtinerea de informatii suplimentare precum $FUNCTION (numele functiei) $PID (id-ul procesului) etc.

Dezavantajul principal este ca nu avem echivalent pentru comenzi care apartin clasei Trace, asadar nu se pot obtine informatii atunci cand aplicatia utilizeaza blocuri release.

In practica se procedeaza astfel: se utilizeaza spatiul de nume Diagnostics atunci cand se doreste ca intodeauna sa se obtina informatii suplimentare despre program, in particular cand stringul de obtinut este complex si implica multe informatii despre variabile. In plus, sunt posibile doar comenzile clasei Trace in configratia Release. Marcajele tracepoints sunt utilizate pentru o analiza rapida a codului in scopul corectarii erorilor semantice.

Depanare in modul break

Facilitatile de depanare oferite de modul break pot fi utilizate prin intermediul toolbarul Debug (care poate fi inclus in toolbar-ul VS prin click dreapta pe toolbar-ul VS, iar apoi selectati Debug) fereastra Breakpoints (meniul Debug -> Windows -> Breakpoints) si o serie de alte ferestre care devin active odata ce s-a intrat in modul break.

Primele patru butoane ale toolbar-ului Debug (Start, Pause, Stop, Restart) permit controlul manual al depanarii in modul break. In afara primului, celelalte sunt inactive pentru un program care nu se executa la momentul respectiv. Odata ce aplicatia este pornita (se apasa butonul Start), ea poate fi intrerupta (butonul Pause), intrandu-se astfel in modul break, sau poate fi parasita complet (butonul Stop) sau restartata (butonul Restart).

Intreruperea aplicatiei cu butonul Pause este unul dintre cele mai simple moduri de a intra in modul break. Insa acesta nu ne permite controlul asupra locului unde aplicatia sa se intrerupa. In acest sens, se utilizeaza breakpoint-urile.

Breakpoints. Un breakpoint reprezinta un punct (marcaj) in codul sursa care declanseaza intrarea automata in modul break.

Un breakpoint poate fi configurat pentru a face urmatoarele: intra in modul break imediat ce breakpoint-ul este atins, intra in modul break daca o expresie boleana este evaluata cu true, intra in modul break daca un breakpoint este atins de un anumit numar de ori, intra in modul break odata ce breakpoint-ul este atins, iar o anumita variabila si-a schinbat valoarea fata de ultima data cand breakpoint-ul a fost atins, afiseaza un text in fereastra Output sau executa un macro.

Toate acestea sunt permise doar in configuratia debug. Daca aplicatia este compilata in configuratia release, toate breakpoint-urile sunt ignorate.

Exista mai multe moduri de a introduce un breakpoint, si anume: urmand aceeasi pasi ca in introducerea unui tracepoint, daca fereastra Breakpoints este activa atunci un simplu click in stanga codului, sau apasati F9.

Utilizand **fereastra Breakpoints** puteti realiza urmatoarele operatii: dezactiva/activa un breakpoint, sterge un breakpoint, edita proprietatile unui breakpoint. Coloanele Condition si Hit Count, afisate de fereastra Breakpoints, sunt cele mai importante. Acestea pot fi editate (click dreapta) pentru a oferi functionalitatea dorita fiecarui breakpoint. Spre exemplu, putem configura un breakpoint in asa fel incat sa intram in modul break daca valMax este mai mare ca 6 prin tastarea expresiei “valMax>6” si selectarea optiunii Is true (coloana Condition).

Metodele Debug.Assert() si Trace.Assert(). Un alt mod de a intra in modul break este de a utiliza una din aceste doua metode. Aceste comenzi pot intrerupe executia programului cu un mesaj precizat de programator. Ele sunt utilizate pentru a testa daca in timpul dezvoltarii aplicatiei lucrurile decurg normal si nu sunt erori semantice. Spre exemplu, pe durata unei aplicatii, o anumita variabila trebuie sa ramana mai mica decat 8. Puteti utiliza metoda Assert() pentru a confirma acest lucru. In acest sens, metoda amintita primeste un parametru de tip bolean. Atata timp cat *valoarea parametrului este true, aplicatia se executa normal*. Daca parametrul isi schimba valoarea in false atunci apare o fereasta care contine, pe langa o serie de informatii privind adresa, functia, etc. si un mesaj (un string sau doua) care reprezinta un alt parametru al metodei Assert(). Avem posibilitatea de a termina aplicatia (Abort), de a intra in modul break (Retry) sau de a continua rularea normala a aplicatiei (Ignore).

Odata ce am intrat in modul break, avem la dispozitie diverse tehnici care permit o analiza detaliata a codului si starii aplicatiei la momentul cand executia a fost intrerupta.

Monitorizarea valoriilor variabilelor Monitorizarea continutului unei variabile este un exemplu simplu care reda utilitatea depanarii in VS. Unul din cele mai simple moduri de a obtine valoarea unei variabile este de duce mouse-ul peste numele acelei variabile. In acelasi mod putem obtine valoarea unei expresii sau valorile elementelor unui vector.

Odata ce aplicatia a intrat in modul break sunt disponibile mai multe ferestre (meniul Debug -> Windows). Ferestrele: **Autos** (contine variabilele in uz si variabilele din precedentele declaratii), **Locals** (toate variabilele din acel domeniu de vizibilitate (sau bloc)), **Watch n**: (variabile si expresii care pot fi incluse la optiunea utilizatorului; n ia valori de la 1 la 4, asadar sunt posibile 4 ferestre).

Fiecare din aceste ferestre se comporta oarecum la fel, cu cateva proprietati particulare, depinde de functia specifica pe care o indeplineste. In general, fiecare fereastra contine o lista de variabile care prezinta informatii despre fiecare variabila:nume, valoare, tip. Variabilele mai complexe, precum tablourile pot fi examinate prin simbolurile + sau -.

Puteti edita continutul unei variabile prin inserarea noii valori in coloana Value. Astfel, se neglijeaza valoarea atribuita variabilei intr-un cod anterior. Aceasta operatiune merita facuta in cazul in care doriti sa incercati diverse scenarii care altfel ar necesita modificarea codului.

Fereastra Watch permite monitorizarea unor variabile la optiunea utilizatorului. Pentru a utiliza aceasta fereastra, scrieti numele variabilei sau expresiei in aceasta fereastra. Un aspect interesant, legat de aceasta fereastra, este faptul ca ea arata care dintre variabile si-a schimbat valoarea intre doua breakpoint-uri (sau de la o secventa de cod la alta daca se utilizeaza unul din butoanele Step Into, Step Over sau Step Out). Noua valoare este scrisa cu rosu.

Executia aplicatiei pas cu pas. Pana in momentul de fata am discutat despre ce se intampla in momentul in care se intra in modul break. Este momentul sa vedem cum putem executa codul programului comanda cu comanda (sau bloc cu bloc), fara a iesi din modul break.

Atunci cand se intra in modul break, apare un cursor in partea stanga a codului care urmeaza a fi executat. La acest moment, putem executa codul programului linie cu linie. In acest sens, utilizam, dupa caz, butonul al saselea (Step Into), al saptelea (Step Over) sau al optulea (Step Out) din toolbarul Debug. Astfel, Step Into executa o comanda si muta cursorul la urmatoarea comanda, Step Over similar butonului anterior, insa nu intra in blocurile interioare (inclusiv blocurile functiilor), Step Out ruleaza pana la sfarsitul blocului si reintra in modul break la instructiunea care urmeaza.

Daca executam aplicatia utilzand aceste butoane, vom observa in ferestrele descrise mai sus cum valorile variabilelor se modificari. Putem sa monitorizam astfel o variabila sau un set de variabile. Pentru codul care prezinta erori semantice, aceasta tehnica poate fi cea mai utila si eficienta.

Fereastra Immediate. Aceasta ferestra (aflata in meniul Debug Windows) permite executia unei comenzi in timp ce aplicatia ruleaza. Cel mai simplu mod de a utiliza aceasta fereastra este de a evalua o expresie. In acest sens, tastati expresia si apasati enter. Rezultatul va fi scris pe linia urmatoare.

Fereastra Call Stack. Aceasta fereastra arata modul in care s-a ajuns in locatia curenta. Adica este aratata metoda curenta f(), metoda g() care a apelat metoda curenta, metoda care a apelat metoda g() si asa mai departe. Aceasta fereasta este utila atunci cand se detecteaza o eroare intrucat putem vedea ce se intampla imediat inaintea erorii. Intrucat multe din erori au loc in blocul unor functii, aceasta fereastra este utila in gasirea sursei erorii.

Tratarea exceptiilor

Exceptii. Clase care reprezinta exceptii

O *exceptie* este o eroare care se produce la momentul executiei. O linie de cod poate sa nu se execute corect din diverse motive: depasire aritmetica, memorie insuficienta, indici in afara intervalului, fisierul din care se citesc sau se scriu datele nu poate fi deschis, etc.

Exemplul de mai jos genereaza exceptie deoarece se incearca impartirea prin zero:

```
class test {
public static void Main()
{   int a = 2, b=0;
    System.Console.WriteLine(a / b);  }  }
```

O aplicatie care si-ar propune sa verifice toate (sau aproape toate) posibilitatile ce pot aparea in executarea unei linii de cod si-ar pierde din claritate, ar fi foarte greu de intretinut, iar mare parte din timp s-ar consuma cu aceste verificari. Realizarea unei astfel de aplicatii este aproape imposibila.

Utilizand sistemul de tratare a exceptiilor din C#, nu este nevoie sa scriem cod pentru a detecta toate aceste posibile caderi. In plus codul se executa mai rapid.

Dintre avantajele introducerii exceptiilor amintim :

- abilitatea de a mentine cod clar si de a intelege logica aplicatiei mai usor;
- posibilitatea de a detecta si localiza bug-uri in cod;

In C# exceptiile sunt reprezentate prin clase. Toate clasele care prezinta exceptii trebuie sa derive din clasa predefinita Exception, care face parte din spatiul de nume System. Din clasa Exception deriva mai multe clase, doua dintre acestea, SystemException si ApplicationException implementeaza doua categorii generale de exceptii definite din limbajul C#: cele generate de motorul de executie si cele generate de programele de aplicatie. De exemplu, codul de mai sus produce o exceptie de tipul DivideByZeroException. Clasa DivideByZeroException deriva din ArithmeticException care la randul ei deriva din clasa SystemException.

Utilizatorul poate defini propriile sale clase care reprezinta exceptii, derivandu-le din clasa ApplicationException.

Notiuni de baza despre tratarea exceptiilor

Tratarea exceptiilor in C# se realizeaza utilizand patru cuvinte cheie: try, catch, throw si finally.

Instructiunile din program care trebuie verificate pentru aparitia exceptiilor sunt incluse in corpul blocului try. Daca pe parcursul executiei unui bloc try se produce o exceptie, atunci aceasta este lansata. Programul poate intercepta exceptia utilizand catch, tratand-o in conformitate cu logica situatiei. Exceptiile de sistem sunt lansate in mod automat de catre motorul de executie din C#. Pentru a lansa manual o exceptie, puteti utiliza cuvantul cheie throw. Codul care trebuie neaparat executat la iesirea dintr-un bloc try trebuie inclus intr-un bloc finally.

La baza tratarii exceptiilor stau constructiile try si catch. Acestea lucreaza in pereche (nu este posibil sa apara try fara catch sau invers). Forma generala a blocurilor try si catch este cea de mai jos:

```
try { // bloc de cod monitorizat pentru detectarea erorilor la executie}
catch(TipExceptie1 exOb){ //rutina de tratare pentru TipExceptie1}
catch(TipExceptie2 exOb){ //rutina de tratare pentru TipExceptie2} …..
catch(TipExceptieN exOb){ //rutina de tratare pentru TipExceptieN}
```

unde TipExceptie reprezinta tipul exceptiei care s-a produs, iar exOb memoreaza valoarea exceptiei (refera obiectul instanta a clasei TipExceptie). Specificarea exOb este optionala. Daca rutina de tratare nu are nevoie de acces la obiectul care reprezinta exceptia, nu este nevoie sa specificam exOb. Mai mult, este optionala specificarea intregii paranteze (TipExceptie exOb). In acest caz, catch are forma: catch {//rutina de tratare a exceptiei}.

Secventa de cod de mai sus functioneaza astfel: Se monitorizeaza codul din interiorul blocului try pentru a detecta posibilele exceptii. Daca o exceptie s-a produs, atunci aceasta este lansata si interceptata de instructiunea catch corespunzatoare. Corpul instructiunii catch care a interceptat exceptia contine secvente de cod pentru tratarea acesteia. O instructiune catch care specifica o exceptie de tip TipExceptie, intercepteaza o exceptie doar daca aceasta reprezinta un obiect instanta a clasei TipExceptie sau un obiect al unei clase derivate din TipExceptie. Spre exemplu, catch(Exception) intercepteaza toate exceptiile deoarece clasa Exception reprezinta clasa de baza pentru toate celelalte clase care reprezinta exceptii. Daca o exceptie este interceptata de o instructiune catch atunci toate celelalte instructiuni catch sunt ignorate. Executia programului continua cu instructiunea care urmeaza dupa catch.

Daca blocul try s-a executat normal atunci toate instructiunile catch sunt ignorate. Programul continua cu prima instructiune dupa ultimul catch.

Exemplul1 (Tratarea exceptiilor)

```
using System;
class ExceptieDemo
{
    public static void Main()
    {
        double[ ] vector = new double[4];
        int a = 1, i = -1;
        do
        {
            try
            {
                i++;
                vector[i] = a / i;
            }
            catch (DivideByZeroException exc)
            {  Console.WriteLine("S-a produs exceptia " + exc.Message);  }
            catch (IndexOutOfRangeException ex)
            {  Console.WriteLine("S-a produs exceptia " + ex.Message);    }
        }
        while (i < 4);
    }
}
```

Rezultat:

S-a produs exceptia Attempted to divide by zero.
S-a produs exceptia Index was outside the bounds of the array.

Programul de mai sus genereaza doua exceptii la rularea sa. O prima exceptie se produce atunci cand variabila intreaga i ia valoarea zero (DivideByZeroException), in timp ce cea de-a doua se produce atunci cand i=4 (IndexOutOfRangeException). Ambele exceptii sunt tratate de instructiunile catch corespunzatoare.

Interceptarea tuturor exceptiilor

Pentru a intercepta toate exceptiile, indiferent de tipul lor, utilizati un catch fara parametru sau, avand in verere ierarhia claselor care reprezinta exceptii, un catch avand ca parametru clasa Exception). Exemplul de mai jos, instructiunea catch intercepteaza atat exceptia DivideByZeroException cat si exceptia IndexOutOfRangeException.

Exemplul 2 (Interceptarea tuturor exceptiilor)

```
using System;
class ExceptieDemo
{
    public static void Main()
    {
        double[ ] vector = new double[4];
        int a = 1, i = -1;
        do
        {
            try
            {
                i++;
                vector[i] = a / i;
            }
            catch
            {  Console.WriteLine("S-a produs o exceptie " );  }
        }
        while (i < 4);
    }
}
```

Rezultat:

S-a produs o exceptie
S-a produs o exceptie

Imbricarea blocurilor try

Este posibil ca un bloc try sa fie continut in interiorul altuia. Exceptiile generate de blocul try interior si care nu sunt interceptate de instructiunile catch asociate acelui bloc sunt transmise blocului try exterior si deci pot fi interceptate de instructiunile catch asociate blocului try exterior. In exemplul de mai jos, exceptia IndexOutOfRangeException este lansata in blocul try interior, insa este interceptata de cel exterior.

Exemplul 3 (Imbricarea blocurilor try)

```
using System;
class ExceptieDemo
{
   public static void Main()
   {
      double[ ] vector = new double[4];
      int a = 1, i = -1;
      try       //exterior
      {   do
         {
            try  //interior
            { i++;  vector[i] = a / i; }
            catch (DivideByZeroException exc)
            { Console.WriteLine("S-a produs exceptia " + exc.Message); }
         }
         while (i < 4);
      }
      catch (IndexOutOfRangeException exc)
      {   Console.WriteLine("S-a produs exceptia " + exc.Message);
         Console.WriteLine("Eroare critica-programul se termina");  }
   }
}
```

Rezultat:

S-a produs exceptia Attempted to divide by zero.

S-a produs exceptia Index was outside the bounds of the array.

Eroare critica-programul se termina

Blocurile try imbricate se utilizeaza pentru a permite tratarea diferentiata a categoriilor diferite de erori. Unele tipuri de erori sunt critice si nu pot fi remediate, in timp ce altele sunt minore si pot fi tratate imediat in corpul instructiunilor catch. Un bloc try exterior intercepteaza o instructiune severa, in timp ce blocurile interioare erori minore care pot fi tratate.

Lansarea unei exceptii

Exemplele precedente intercepteaza exceptii generate automat de C#.

Este insa posibil sa lansam manual o exceptie, utilizand instructiunea throw a carei forma generala este:

throw exOb;

exOb trebuie sa fie un obiect al carui tip este o clasa derivata din Exception.

In programul alaturat este lansata o exceptie de tipul InsufficientMemoryException

Remarcati modul in care a fost produsa exceptia. Intrucat o exceptie este un obiect, trebuie utilizat operatorul new pentru a crea obiectul instanta al clasei InsufficientMemoryException.

Exemplul 4 (Lansarea manuala a unei exceptii)

```
using System;

class ThrowDemo
{
   public static void Main()
{
      try
   {
      Console.WriteLine("Inainte de throw");
      throw new InsufficientMemoryException();
   }
   catch (InsufficientMemoryException)
   {
      Console.WriteLine("Exceptie
    interceptata");
   }
}
}
```

Rezultat:

Inainte de throw
Exceptie interceptata

Relansarea unei exceptii

O exceptie interceptata de o instructiune catch poate fi relansata, cu scopul de a fi interceptata de un bloc catch exterior. Exceptiile sunt relansate pentru a fi tratate de mai multe rutine de tratare. De exemplu, o prima rutina poate trata un aspect al exceptiei, in timp ce celelalte aspecte sunt tratate de o rutina aflata intr-un alt bloc catch. Pentru a relansa o exceptie se utilizeaza instructiunea throw fara a specifica vreo exceptie, ca in exemplul de mai jos. Exceptia relansata de blocul interior va fi interceptata de blocul exterior.

Exemplul 5 (Relansarea unei exceptii)

```
using System;
class ExceptieDemo
{    public static void Main()
    {   double[ ] vector = new double[4];
        int a = 1, i = -1;
        try       //exterior
        {   do
            {   try  //interior
                { i++; vector[i] = a / i; }
                catch (DivideByZeroException exc)
                { Console.WriteLine("S-a produs exceptia " + exc.Message); }
                catch (IndexOutOfRangeException exc)
                {   Console.WriteLine("Exceptia: " + exc.Message+ " In blocul try interior ");
                    throw; }  //relansarea exceptiei
            }
            while (i < 4);
        }
        catch (IndexOutOfRangeException exc)
        {    Console.WriteLine("Exceptia: " + exc.Message+" Relansata");   }
    }
}
```

Rezultat:

S-a produs exceptia Attempted to divide by zero.
Exceptia: Index was outside the bounds of the array. In blocul try interior
Exceptia: Index was outside the bounds of the array. Relansata

Utilizarea lui finally

In unele situatii este necesar ca un bloc de instructiuni sa se execute la terminarea unui bloc try/catch. Spre exemplu, o exceptie care se produce in program cauzeaza terminarea prematura a unei metode care a deschis un fisier sau o conexiune la retea care trebuie inchisa. Limbajul C# pune la dispozitie o modalitate convenabila de tratare a acestor probleme prin intermediul blocului finally.

Pentru a specifica un bloc de cod care trebuie sa se execute neaparat la iesirea dintr-o secventa try/catch, fie ca blocul try s-a executat normal fie ca s-a terminat datorita unei exceptii, se utilizeaza un bloc finally. Forma generala a unui bloc try/catch care include un bloc finally este:

```
try { // bloc de cod monitorizat pentru detectarea erorilor la executie}
catch(TipExceptie1 exOb){ //rutina de tratare pentru TipExceptie1}
catch(TipExceptie2 exOb){ //rutina de tratare pentru TipExceptie2}
 ………………………………………………………………………..
catch(TipExceptieN exOb){ //rutina de tratare pentru TipExceptieN}
finally{ //bloc final}
```

Clasa Exception

Asa cum am vazut toate exceptiile sunt derivate din clasa Exception. Aceasta clasa defineste mai multi constructori, metode si cateva proprietati. Trei dintre cele mai interesante proprietati sunt Message, StackTrace si TargetSite. Toate trei sunt accesibile numai la citire. Message contine un string care descrie natura erorii, StackTrace contine un alt string care prezinta stiva apelurilor care a condus la aparitia exceptiei. TargetSite intoarce un obiect care specifica metoda care a generat exceptia. Dintre metodele clasei Exception, amintim metoda ToString() care intoarce un string care descrie exceptia. Metoda ToString() este apelata automat atunci cand exceptia este afisata prin intermediul lui WriteLine().

Exemplul 6 (Utilizarea proprietatilor clasei Exception)

```
using System;
class ExceptieDemo
{    public static void Main()
    {   double[ ] vector = new double[5];
        int a = 1, i = -1;
        do
        {   try
            {   i++;        vector[i] = a / i;       }
            catch(DivideByZeroException exc)
            {   Console.WriteLine("S-a produs o exceptie ");
                Console.WriteLine("Mesajul standard este: \n" + exc);
                Console.WriteLine("\nStiva de apeluri: " + exc.StackTrace);
                Console.WriteLine("Mesajul: " + exc.Message);
                Console.WriteLine("Metoda care a generat exceptia: " + exc.TargetSite);     }
        }
        while (i < 4);
    }
}
```

Rezultat:

```
S-a produs o exceptie
Mesajul standard este: System.DivideByZeroException: Attempted to divide by zero.
   at ExceptieDemo.Main() in H:\C#appl\Project14\Project14\CodeFile1.cs:line 16
Stiva de apeluri:    at ExceptieDemo.Main() in H:\C#appl\Project14\Project14\CodeFile1.cs:line 16
Mesajul: Attempted to divide by zero.
Metoda care a generat exceptia: Void Main()
```

Clase derivate din Exception

Erorile cel mai frecvent intalnite sunt tratate utilizand exceptiile predefinite in C#. Insa, mecanismul de tratare a exceptiilor nu se limiteaza doar la acestea. Utilizatorul poate defini exceptii personalizate pentru tratarea erorilor specifice care pot aparea in programele sale.

Pentru a crea un nou tip de exceptii este suficient sa definiti o clasa derivata din Exception. Ca regula generala (insa nu obligatorie), exceptiile definite de utilizator trebuie sa fie derivate din clasa ApplicationException, acceasta fiind radacina ierarhiei rezervate pentru exceptiile generate de aplicatii. Clasele derivate de utilizator nu trebuie sa implementeze nimic, simpla lor apartenenta la ierarhia de tipuri permite utilizarea lor ca exceptii.

Clasele create de utilizator dispun, prin mostenire, de proprietatile si metodele definite de clasa Exception. Unul sau mai multi membrii ai clasei Exception pot fi extinsi in clasele reprezentand exceptii pe care utilizatorul le poate crea.

Spre exemplu, programul urmator creaza o exceptie numita NonIntResultException. Aceasta exceptie este lansata in program atunci cand impartirea a doua valori intregi produce un rezultat cu o parte fractionara. Clasa NonIntResultException defineste doi constructori standard si redefineste metoda ToString().

Exemplu 7 (Clasa derivata din Exception)

```
using System;
class NonIntResultException: ApplicationException
{
   public NonIntResultException()
      : base()
   { }
   public NonIntResultException(string str)
      : base(str)
   { }
   public override string ToString()
  { return Message;}

}
class ExceptieDemo
{
   public static void Main()
   {   int[ ] numarator = {2, -3, -2, 3, 47, 0, 23, 34, 23, 345, 48, 38, 34};
      int[] numitor = { 4, 0, -6, 3, 47, 0, 23, 34, 23 };
      for (int i = 0; i < numarator.Length; i++)
      {   try
         {   int a;
            a = numarator[i] / numitor[i];
            if (numarator[i] % numitor[i] != 0)
            { throw new NonIntResultException("Rezultatul "+ numarator[i]+ "/"+ numitor[i]+ " nu este intreg"); }
            Console.WriteLine("{0}/{1} este {2}", numarator[i], numitor[i], a);
         }
         catch(DivideByZeroException)
         { Console.WriteLine("Nu putem imparti prin zero"); }
         catch (IndexOutOfRangeException)
         { Console.WriteLine("Elementul nu exista in tabel"); }
         catch(NonIntResultException exc)
         { Console.WriteLine(exc); }
      }
   }
}
```

Rezultat:

Rezultatul 2/4 nu este intreg
Nu putem imparti prin zero
Rezultatul -2/-6 nu este intreg
3/3 este 1
47/47 este 1
Nu putem imparti prin zero
23/23 este 1
34/34 este 1
23/23 este 1
Elementul nu exista in tabel
Elementul nu exista in tabel
Elementul nu exista in tabel
Elementul nu exista in tabel

Interceptarea exceptiilor derivate

Daca trebuie sa interceptati exceptii avand ca tipuri atat o clasa de baza cat si o clasa derivata atunci clasa derivata trebuie sa fie prima in secventa de instructiuni catch. In caz contrar, instructiunea catch asociata clasei de baza va intercepta si exceptiile clasei derivate, conducand la existenta codului inaccesibil (codul din instructiunea catch asociata clasei derivate nu se va executa niciodata). *In C#, prezenta instructiunii catch inaccesibile este considerata eroare*. Programul de mai jos va genera eroare la compilare.

```
using System;
class NonIntResultException: ApplicationException
{    public NonIntResultException()
        : base()    { }
    public NonIntResultException(string str)
        : base(str)   { }
    public override string ToString()
   { return Message;}
}
class ExceptieDemo
{    public static void Main()
    {   int[ ] numarator = {2, -3, -2, 3, 47, 0, 23, 34, 23, 345, 48, 38, 34};
        int[] numitor = { 4, 0, -6, 3, 47, 0, 23, 34, 23 };
        for (int i = 0; i < numarator.Length; i++)
        {   try
            {   int a;   a = numarator[i] / numitor[i];
                if (numarator[i] % numitor[i] != 0)
                { throw new NonIntResultException("Rezultatul "+ numarator[i]+ "/"+ numitor[i]+ " nu este intreg"); }
                Console.WriteLine("{0}/{1} este {2}", numarator[i], numitor[i], a);
            }
            catch(DivideByZeroException)
            { Console.WriteLine("Nu putem imparti prin zero"); }
            catch (Exception)                                          //ordinea instructiunilor catch conteaza
            { Console.WriteLine("Elementul nu exista in tabel"); }  //programul genereaza o eroare la compilare
            catch(NonIntResultException exc)
            { Console.WriteLine(exc); }
        }          }      }
```

Operatii de intrare-iesire

Sistemul de intrare-iesire din C# este constituit ca o ierarhie de clase. Intrucat limbajul C# utilizeaza clasele arhitecturii .NET, discutia despre intrari-iesiri in C# este deci o discutie despre sistemul de intrare-iesire al arhitecturii .NET in general.

Ne propunem sa trecem in revista o serie de clase utilizate pentru crearea si gestionarea fisierelor, citirea din si scrirerea in fisiere, precum si ideile principale in care limbajul C# implementeaza intrarile si iesirile.

Conceptul de stream. Programele C# efectueaza operatii de intrare-iesire prin intermediul stream-urilor. Un stream este o entitate abstracta care fie produce, fie consuma informatii. Sistemul de intrare-iesire din C# asociaza stream-urile cu dispozitivele fizice (de regula discul magnetic) utilizate efectiv. Toate stream-urile se comporta la fel, chiar daca dispozitivele fizice nu sunt aceleasi. De regula, utilizam aceleasi metode pentru a scrie atat in consola cat si intr-un fisier de pe disc sau intr-o retea.

Putem clasifica streamurile in streamuri de iesire (Output stream) si streamuri de intrare (Input stream). Streamurile Output sunt utilizate atunci cand datele sunt scrise intr-un dispozitiv extern, care poate fi: in fisier de pe discul fizic, o imprimanta, o retea sau un alt program. Streamurile Input sunt folosite pentru a citi date. Un stream Input poate proveni din orice sursa: tastatura, fisier de pe disc, retea, etc.

In cele ce urmeaza ne vom axa pe studiul operatiilor de intrare/iesire care utilizeaza fisiere. Conceptele aplicate pentru citirea sau scrierea fisierelor se aplica majoritatii dispozitive fizice. Astfel, acestea pot fi aplicate diverselor situatii care pot aparea in practica.

Clase pentru operatii de intrare-iesire. Spatiul de nume System.IO contine majoritatea claselor pentru lucrul cu fisiere. Vom analiza in cele ce urmeaza urmatoarele clase:

File	O clasa statica care expune numeroase metode statice pentru copierea, steregerea, crearea fisierelor.
Directory	O clasa statica care expune numeroase metode statice pentru copierea, steregerea, crearea directoarelor.
Path	O clasa care realizeaza operatii asupra stringurilor care reprezinta numele sau calea catre fisiere sau directoare.
FileInfo	Un fisier este reprezentat prin intermediul unui obiect de tipul acestei clase. Obiectul beneficiaza de o serie de metode pentru manipularea fisierului respectiv.
DirectoryInfo	Similar clasei FileInfo, insa obiectul reprezinta un director.
FileSystemInfo	Serveste drept clasa de baza pentru FileInfo si DirectoryInfo. Utilizant conceptul de polimorfism, creaza facilitatea de a lucra simultan cu fisiere si directoare.
FileStream	Un strem de tipul acestei clase permite scrierea intr-un sau citirea dintr-un fisier.
Stream	O clasa abstracta care sta la baza implementarii conceptului de stream in C#. Este clasa de baza pentru toate celelalte clase care reprezinta stream-uri, inclusiv FileStream.

StreamReader	Citeste caractere dintr-un stream. Un stream de acest tip poate fi creat prin impachetarea unui stream de tipul FileStream.
StreamWriter	Scrie caractere intr-un stream. La fel ca in cazul clasei de mai sus, stream de acest tip poate fi creat prin impachetarea unui stream de tipul FileStream.
FileSystemWatcher	Se utilizeaza pentru a monitoriza fisiere si directoare si expune evenimente pe care aplicatia utilizatorului le poate intercepta atunci cand au loc modificari in aceste locatii.

Vom aborda problema serializarii obiectelor utilizand spatiul de nume System.Runtime.Serialization si spatiile de nume pe care le contine. Mai precis, vom analiza clasa BinaryFormatter din spatiul de nume System.Runtime.Serialization.Formatters.Binary, care permite serializarea obiectelor intr-un stream ca date binare si deserializarea lor.

Clasele File si Directory. Aceste doua clase expun numeroase metode statice pentru manipularea fisierelor si directoarelor. Aceste metode fac posibile o serie de operatii asupra fisierelor si directoarelor precum si crearea unor streamuri FileStream. Toate metodele sunt statice, asadar acestea sunt apelate fara a crea instante ale clasei File sau Directory.

Cateva dintre cele mai utile metode ale clasei File sunt:

Copy()	Copiaza un fisier dintr-o locatie sursa intr-o locatie destinatie;
Create()	Creaza un fisier in directorul specificat (sau utilizand calea specificata);
Delete()	Sterge un fisier;
Open()	Returneaza un stream FileStream corespunzator stringului (care reprezinta calea) specificat;
Move()	Muta un fisier intr-o noua locatie. Fisierului i se poate specifica un alt nume in noua locatie.

Cateva metode utile ale clasei Directory sunt:

CreateDirectory()	Creaza un director utilizand calea specificata;
Delete( )	Sterge directorul specificat si toate fisierele continute de acesta;
GetDirectories()	Returneaza un tablou de tip string care contine numele directoarelor aflate in directorul specificat;
GetFiles()	Returneaza un tablou de tip string care contine numele fisierelor aflate in directorul specificat;
GetFileSystemEntries()	Returneaza un tablou de tip string care contine numele fisierelor si directoarelor aflate in directorul specificat;
Move()	Muta un director intr-o noua locatie. Directorului i se poate specifica un alt nume in noua locatie.

Exemplu: Programul returneaza fisierele sau/si directoarele continute pe unitatea D

```
using System;
using System.IO;
class Program
{
    public static void Main()
    {
        string str;
        char c;
        string[] tablou;
        do{
            Console.WriteLine(@"Type:
    f) for files;
    d) for directories
    e) for files and directories");
        str = Console.ReadLine();
        }
        while((str!="f") ^ (str!="d") ^(str!="e"));
        c=char.Parse(str);
switch(c)
    {
        case 'f':
            tablou=Directory.GetFiles(@"D:\");
            foreach (string s in tablou)
                Console.WriteLine(s);
            break;
        case 'd':
            tablou=Directory.GetDirectories(@"D:\");
            foreach (string s in tablou)
                Console.WriteLine(s);
            break;
        case 'e':
            tablou=
        Directory.GetFileSystemEntries(@"D:\");
            foreach (string s in tablou)
                Console.WriteLine(s);
            break;
    }
  Console.ReadKey();
  }
}
```

Clasa FileInfo. Contrar clasei File, clasa FileInfo nu este statica si nu contine metode statice. Asadar, clasa este utila doar cand este instantiata. Un obiect de tipul FileInfo reprezinta un fisier de pe disc sau dintr-o retea si poate fi creat doar prin furnizarea caii catre acel fisier. Spre exemplu, daca pe unitatea C: \ se afla fisierul Log.tex atunci instructiunea FileInfo unFisier=new FileInfo(@"C:\Log.tex"); creaza un obiect FileInfo care reprezinta acest fisier.

Multe dintre metodele expuse de FileInfo sunt similare celor din clasa File, insa deoarece clasa File este statica, la apelarea fiecarei metode este necesara specificarea unui paramentru de tip string care sa specifice locatia fisierului. Spre exemplu: instructiunile de mai jos realizeaza acelasi lucru:

a)
```
FileInfo unFisier=new FileInfo(@"C:\Log.tex");
if (unFisier.Exists)
Console.WriteLine("Fisierul exista");
```

b)
```
if(File.Exists("C:\\Log.tex"))
 Console.WriteLine("Fisierul exista");
```

In majoritatea cazurilor, nu are importanta care tehnica este utilizata pentru manipularea fisierelor, insa se poate utiliza urmatorul criteriu pentru a decide care este mai potrivita: -are sens sa utilizam metodele clasei File daca apelam o metoda o singura data. Un singur apel se executa rapid intrucat nu se trece in prealabil prin procesul de instantiere a unui obiect; -daca aplicatia realizeaza mai multe operatii asupra unui fisier atunci are sens instantierea unui obiect si utilizarea metodelor acestuia. Se salveaza timp intrucat obiectul refera deja fisierul corect, in timp ce clasa statica trebuie sa il gaseasca de fiecare data.

Clasa FileInfo expune de asemenea si proprietati care pot fi utilizate pentru manipularea fisierului referit. Multe dintre aceste proprietati sunt mostenite de la clasa FileSystemInfo, si deci se aplica atat clasei FileInfo cat si clasei DirectoryInfo. Proprietatile clasei FileSystemInfo sunt:

Attributes	Citeste sau scrie atributele fisierului sau directorului curent utilizand enumerarea FileAttributes;
CreationTimeUtc CreationTime	Citeste si scrie data si ora crearii fisierului sau directorului curent, in timp universal (UTC-coordinated universal time) sau nu;
Extension	Intoarce extensia fisierului, accesibila doar la citire;
Exists	Motoda abstracta acesibila doar la citire. Este implementata atat de FileInfo cat si de DirectoryInfo si determina daca fisierul sau directorul exista;
FullName	Proprietate virtuala accesibila doar la citire. Intoarce calea pana la fiser sau director;
LastAccessTimeUtc LastAccessTime	Citeste si scrie data si ora cand fisierul (directorul) a fost accesat (in UTC si non-UTC);
LastWriteTimeUtc LastWriteTime	Citeste si scrie data si ora cand s-a scris in fisier (sau director) (in UTC si non-UTC);
Name	Proprietate abstracta accesibila doar la citire. Intoarce calea pana la fiser sau director. Este implementata atat de FileInfo cat si de DirectoryInfo.

Proprietatile specifice clasei FileInfo sunt:

Directory	Intoarce un obiect de tipul DirectoryInfo care reprezinta directorul care contine fisierul curent. Proprietate accesibila la citire;
DirectoryName	Intoarce un string care reprezinta calea pana la directorul care contine fisierul curent. Proprietate accesibila la citire;
IsReadOnly	Determina daca fisierul este sau nu accesibil doar la citire;
Length	Intoarce o valoare de tip long care reprezinta dimensiunea fisierului in octeti. Proprietate accesibila la citire;

Un obiect FileInfo nu reprezinta un stream. Pentru a scrie intr-un sau citi dintr-un fisier trebuie creat un obiect de tipul Stream. Clasa FileInfo poate fi utila in acest sens intrucat expune cateva metode care returneaza obiecte Stream (in fapt obiecte de tipul unor clase derivate din clasa Stream). Vom prezenta aceste metode dupa descrierea streamurilor.

Clasa DirectoryInfo. Clasa DirectoryInfo se utilizeaza ca si clasa FileInfo. Cand este instantiata, obiectul creat reprezinta un director. Multe dintre metode sale sunt metode duplicat ale clasei Directory. Criteriul de alegere dintre metodele File ori FileInfo se aplica la fel si in acest caz.

Majoritatea proprietatilor sunt mostenite de la clasa FileSystemInfo. Doua dintre proprietati sunt specifice:

Parent	Intoarce un obiect DirectoryInfo reprezentant directorul care contine directorul curent. Proprietate acesibila la citire;

Root Intoarce un obiect DirectoryInfo reprezentand unitatea directorului curect. Spre exemplu C:\ Proprietate accesibila la citire.

Exercitiu: rescrieti exemplul anterior folosind metodele si proprietatile claselor FileInfo si DirectoryInfo.

Clasa FileStream. Un obiect FileStream reprezinta un stream care indica spre un anumit fisier de pe disc sau dintr-o retea. Desi, clasa expune mai multe metode pentru citirea sau scrierea octetilor dintr-un sau intr-un fisier, de cele mai multe ori vom utiliza un obiect StreamReader sau StreamWriter pentru a realiza aceste operatii. Aceasta deoarece clasa FileStream opereaza cu octeti (bytes) sau tablouri de octeti, in timp ce clasele StreamReader si StreamWriter opereaza cu date caracter. Este mai convenabil lucrul cu date caracter, insa anumite operatii, spre exemplu accesul unor date aflate undeva la mijlocul fisierului, pot fi realizate doar de un obiect FileStream.

Un obiect FileStream poate fi creat in mai multe moduri. Constructorul are mai multe versiuni supraincarcate, cea mai simpla forma fiind:

```
FileStream unfisier = new FileStream(numeFisier, FileMode.<Member>);
```

unde enumerarea FileMode are mai multi membrii care specifica modul cum fisierul este deschis sau creat. O alta versiune, des intrebuintata este:

```
FileStream unfisier = new FileStream(numeFisier, FileMode.<Member>, FileAccess.<Member>);
```

unde enumerarea FileAccess specifica scopul streamului. Membrii acestei enumerari sunt: Read (deschide fisierul pentru citire), Write (deschide fisierul pentru scriere), ReadWrite (deschide fisierul pentru citire si scriere).

Daca se incearca realizarea unei alte operatii decat cea specificata de enumerarea FileAccess atunci va fi lansata o exceptie.

In prima versiune a constructorului, cea care nu contine parametrul FileAccess, este utilizata valoarea default FileAccess.ReadWrite.

Enumerarea FileMode, impreuna cu o descrire a comportarii membrilor acesteia atat in cazul in care fisierul exista sau nu, este prezentata in tabelul:

Membru	Fisierul exista	Fisierul nu exista
Append	Fisierul este deschis cu stream-ul pozitionat la sfarsitul fisierului. Poate fi utilizat numai in conjunctie FileAccess.Write.	Este creat un fisier. Poate fi utilizat in conjunctie cu cu FileAccess.Write.
Create	Fisierul este distrus si un nou fisier este creat in locul acestuia.	Este creat un nou fisier.
CreateNew	Este lansata o exceptie.	Este creat un nou fisier.
Open	Fisierul este deschis cu stream-ul pozitionat la inceputul fisierului.	Este lansata o exceptie.
OpenOrCreate	Fisierul este deschis cu stream-ul pozitionat la inceputul fisierului.	Este creat un nou fisier.
Truncate	Fisierul este deschis si continutul este sters. Stream-ul este pozitionat la inceputul fisierului. Este retinuta data de creare a fisierului original.	Este lansata o exceptie.

Exemplu: (Utilizarea clasei FileStream. Programul copiaza un fisier)

```
using System;
using System.IO;
class Scrie_Octeti
{
    public static void Main()
   {
   if (File.Exists(@"D:\Curs6.ppt"))
   { try
        {
         int i;
         FileStream fin, fout;
         fin = new FileStream(@"D:\Curs6.ppt",
                                FileMode.Open);
         fout = new FileStream(@"D:\Curs6duplicat.ppt“
                                , FileMode.OpenOrCreate);
do {
         i = fin.ReadByte();
         if (i != -1)
           { fout.WriteByte((byte)i); }
       }
 while (i != -1);
           fin.Close();
            fout.Close();
            }
   catch (IOException exc)
     {
       Console.WriteLine(exc.Message + "Nu poate
       deschide sau accesa unul dintre fisiere");
                return;
            }
        }
    }
}
```

Ambele clase File si FileInfo expun metodele OpenRead() si OpenWrite() care permit crearea de obiecte FileStream. Prima metoda deschide un fisier doar pentru citire, iar cea de-a doua metoda doar pentru scriere. Aceste metode ofera de fapt shortcut-uri incat nu este necesara precizarea parametrilor necesari constructorului clasei FileStream. De exemplu, codul de mai jos deschide fisierul "Data.txt" pentru citire:

```
FileStream unFisier= File.OpenRead("Data.txt");
```

Un rezultat similar se obtine astfel:

```
FileInfo unFisierInfo=new FileInfo("Data.txt");
FileStream unFisier=unFisierInfo.OpenRead();
```

Pozitia in fisier. Clasa FileStream mentine un pointer intern care indica o locatie in fisier unde se va produce urmatoarea operatie de citire sau de scriere. In cele mai multe cazuri, cand un fisier este deschis, pointerul indica inceputul fisierului, insa aceasta pozitie poate fi modificata. Acest fapt permite unei aplicatii sa citeasca sau sa scrie oriunde in fisier.

Metoda care implementeaza aceasta functionalitate este metoda Seek(), care are doi parametrii. Primul parametru, un parametru de tip long, specifica cu cati octeti (bytes) trebuie deplasat pointerul, iar al doilea parametru este enumerarea SeekOrigin care contine trei valori: Begin, Current si End.

Spre exemplu, comanda unfisier.Seek(3, SeekOrigin.Begin) muta pointerul 3 octeti inainte fata de pozitia initiala, sau unfisier.Seek(-5, SeekOrigin.End) muta pointerul 5 pozitii inapoi fata de pozitia de sfarsit a fisierului.

Clasele StreamReader si StreamWriter acceseaza fisierele secvential si nu permit manipularea pointerului in acest fel.

Citirea datelor. Citirea datelor utilizand clasa FileStream nu se face la fel de usor ca in cazul citirii cu ajutorul clasei StreamReader. Aceasta deoarece clasa FileStream lucreaza cu octeti in forma bruta, neprelucrati. Acest fapt permite clasei FileStream o mare flexibilitate in citirea oricaror fisiere, precum imagini, muzica, video, etc. Costul acestei flexibilitati este acela ca nu se poate utiliza un obiect FileStrem pentru a citi date in mod direct si a le converti intr-un string asa cum se poate face cu StreamReader. Cu toate acestea, exista o serie de clase care convertesc tablourile de octeti in tablouri de caractere si viceversa.

Metoda ReadByte() a fost utilizata in exemplul anterior.

O alta metoda a clasei FileStream este Read() care citeste date dintr-un fisier si ii scrie intr-un tablou de tip byte. Metoda intoarce un int care reprezinta numarul de octeti cititi din stream si are trei parametrii. Primul paramentru reprezinta un tablou de tip byte in care vor fi scrise datele, al doilea parametru de tip int specifica pozitia elementului din tablou unde va incepe scrierea (de regula acesta este 0 daca se incepe scrierea la primul element din tablou), iar al treilea parametru indica numarul de octeti cititi din fisier.

Urmatoarea aplicatie citeste 200 de octeti din fisierul Program.cs, incepand de la pozitia 113. Octetii sunt decodificati si convertiti in caractere, utilizand clase din spatiul de nume System.Text;

Scrierea datelor. Procesul de scrire a datelor este similar, trebuie creat un tablou de tip byte. In acest sens, cream un tablou de caractere, apoi sa il codificam intr-un tablou byte si in final il scriem in fisier. Vezi exemplul scrierea datelor intr-un fisier.

Exemplu: Citirea datelor dintr-un fisier

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace CitesteFisier
{
    class Program
    {
        static void Main(string[] args)
        {
            byte[] byData = new byte[200];
            char[] charData = new Char[200];
            try
            {
                FileStream aFile = new
        FileStream("../../Program.cs", FileMode.Open);
                aFile.Seek(113, SeekOrigin.Begin);
                aFile.Read(byData, 0, 200);
            }
catch (IOException e)
    {
    Console.WriteLine("A fost lansata o
                exceptie IO!");
    Console.WriteLine(e.ToString());
    Console.ReadKey();
            return;
    }
  Decoder d = Encoding.UTF8.GetDecoder();
  d.GetChars(byData, 0, byData.Length,
                charData, 0);
  Console.WriteLine(charData);
  Console.ReadKey();
    }
  }
}
```

Exemplu: Scrierea datelor intr-un fisier

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace ScrieInFisier
{
    class Program
    {
        static void Main(string[] args)
        {
            byte[] byData;
            char[] charData;
            try
            {
                FileStream unFisier = new FileStream("Temp.txt",
    FileMode.Create);
                charData = "Un string pe care il scriem in
    fisier.".ToCharArray();
                byData = new byte[charData.Length];
                Encoder e = Encoding.UTF8.GetEncoder();
                e.GetBytes(charData, 0, charData.Length, byData,
    0, true);

// Mutam pointerul la inceputul fisierului.
                unFisier.Seek(0, SeekOrigin.Begin);
                unFisier.Write(byData, 0,
    byData.Length);
            }
            catch (IOException ex)
            {
                Console.WriteLine("S-a produs o
    exceptie IO!");
                Console.WriteLine(ex.ToString());
                Console.ReadKey();
                return;
            }
        }
    }
}
```

Clasele StreamReader si StreamWriter. A lucra cu tablouri de octeti nu este intodeauna placut. O cale mai simpla, odata ce este creat un stream de tipul FileStream, este de a-l impacheta intr-un stream de caractere.

Pentru a impacheta un stream octet intr-un stream caracter trebuie sa utilizati subclasele StreamReader si StreamWriter ale claselor abstracte: TextReader si TextWriter. Odata ce stream-ul este impachetat se pot utiliza metodele acestor doua clase (StreamReader si StreamWriter) pentru a manipula fisierul. Daca nu este necesar schimbarea pozitiei pointerului in interiorul fisierului atunci aceste clase fac lucrul mult mai usor.

Clasa StreamReader contine mai multe metode, dintre care cele mai importante sunt Read(), ReadLine(). Clasa StringWriter contine, (pe langa alti membri) metodele Write() si WriteLine(). Acestea de utilizeaza la fel ca in cazul metodelor similare puse la dispozitie de clasa Console.

Exista mai multe moduri de a crea un obiect StreamReader (sau StreamWriter). Daca deja exista un stream FileStream atunci pentru a crea un StreamWriter se proceaza astfel:

```
FileStream unFisier= new FileStream("Log.tex", FileMode.CreateNew);
StreamWriter sw=new StreamWriter(unFisier);
```

Un obiect StreamWriter poate fi creat direct, utilizand un fisier:

```
StreamWriter sw=new StreamWriter("Log.txt", true);
```

Acest constructor are ca parametrii calea pana la fisier si un parametru bolean care daca este true atunci fisierul este deschis, iar datele continute de acesta sunt retinute. Daca fisierul nu exista atunci se creaza un nou fisier. Daca parametrul este false atunci daca fisierul exista acesta este deschis si continutul sters, iar daca nu exista atunci se creaza un fisier.

Exemplu (Utilizarea clasei StreamWriter)

```
using System;     using System.IO;
class Scrie_Caracter_Octeti {
    public static void Main()    {
        try  {
            string str;
            FileStream fout;
            fout = new FileStream("fisier.txt", FileMode.Create);
            StreamWriter fstr_out = new StreamWriter(fout);
            Console.WriteLine("Introduceti textul fisierului. (Daca doriti sa terminati tastati: stop ");
            do
            {
                str = Console.ReadLine();
                if (str != "stop")
                {
                    str = str + "\r\n";
                    try
                    { fstr_out.Write(str); }
                    catch (IOException exc)
                    {  Console.WriteLine(exc.Message + "Eroare la scrierea in fisier");   return;    }
                }
            }
            while (str != "stop");
            fstr_out.Close();
        }
        catch (IOException exc)
        {
            Console.WriteLine(exc.Message + "Nu poate deschide sau accesa fisierul");
            return;
        }
    }
}
```

Exemplu (Utilizarea Clasei StreamReader)

```
using System;
using System.IO;
class Citeste_Octeti_Caracter
{
   public static void Main()
   {
      string str;
      try
      {
         FileStream fin = new FileStream(“fisier.txt", FileMode.Open);

         StreamReader fstr_in = new StreamReader(fin);
         while ((str = fstr_in.ReadLine()) != null)
         {
            Console.WriteLine(str);
         }
         fstr_in.Close();
      }
      catch (IOException exc)
      {
         Console.WriteLine(exc.Message + "Fisierul nu exista sau nu poate fi accesat");
         return;
      }
   }
}
```

Clasa Stream. Clasa Stream, din spatiul de nume System.IO, sta la baza implementarii conceptului de stream in C#. Clasa Stream reprezinta un stream octet si este clasa de baza pentru toate celelalte clase care reprezinta stream-uri. Aceasta clasa este abstracta, deci nu puteti crea instante ale acestei clase. Ea defineste mai multe metode atat pentru citirea cat si pentru scrierea datelor care sunt implementate de clasele derivate. Nu toate stream-urile implementeaza ambele categorii de date. Unele stream-uri pe care le creati sunt accesibile la scriere, altele la citire.. Dintre metodele clasei stream amintim: void Close(), void Flush(), int ReadByte(), long Seek(), void WriteByte(), etc. intre proprietati amintim: bool CanRead, bool CanSeek, bool CanWrite, etc.

Serializarea obiectelor. Aplicatiile pe care le cream necesita de multe ori stocarea datelor pe hard disc. In exemplele anterioare fisierele au fost construite octet cu octet (sau caracter cu caracter). De multe ori, aceasta cale nu este cea mai convenabila. Uneori este mai bine ca datele sa fie stocate in forma in care se gasesc, adica ca obiecte.

Arhitectura .NET ofera acea infrastructura pentru serializarea obiectelor in spatiile de nume System.Runtime.Serialization si System.Runtime.Serialization.Formatters, prin clasele specifice care sunt puse la dispozitie. Sunt posibile doua implemetari:

System.Runtime.Serialization.Formatters.Binary Acest spatiu de nume contine clasa BinaryFormatter care este capabila sa serializeze obiecte in date binare si viceversa.

System.Runtime.Serialization.Formatters.Soap Acest spatiu de nume contine clasa SoapFormatter care este capabila sa serializeze obiecte in format SOAP pentru date XML, si viceversa.

In cele ce urmeaza vom aborda serializarea obiectelor utilizand BinaryFormatter. In fapt, ambele clase BinaryFormatter si SoapFormatter implementeaza interfata IFormatter care furnizeaza metodele:

void Serialize(Stream stream, object source)	Serializeaza source in stream.
object Deserialize(Stream stream)	Deserializeaza datele din stream si returneaza obiectul rezultat.

Serializarea utilizand BinaryFormatter se poate face simplu, spre exemplu:

```
IFormatter s = new BinaryFormatter();
s.Serialize(streamulMeu, obiectulMeu);
```

Deserializarea este de asemenea simpla:

```
IFormatter s = new BinaryFormatter();
TipObiect noulMeuObiect= s.Deserialize(streamulMeu) as TipObiect;
```

Aceste secvente de cod sunt valide in majoritatea circumstantelor.

Urmatorul exemplu arata cum se procedeaza in practica.

a) Executati programul. Veti observa ca apare o eroare la serializare.

b) Stergeti comentariul din fata atributului [Serializable]. Rulati din nou programul si observati diferenta.

c) De asemenea observati ca stringul Observatii nu a fost serializat intrucat atributul sau este [Nonserialized]

```
using System;   using System.Collections.Generic;
using System.Collections;   using System.Linq;
using System.Text;    using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace Magazin
{
    //[Serializable]
    public class Produs
    {  public long Id;  public string Nume;   public double Pret;
   [NonSerialized]  string Observatii;
        public Produs(long id, string nume, double pret, string observatii)
        {
            Id = id;
            Nume = nume;
            Pret = pret;
            Observatii = observatii;
        }
        public override string ToString()
        {
            return string.Format("{0}: {1} ({2:#.##} lei) {3}", Id, Nume, Pret,  Observatii);
        }
    }
```

```
class Program
    {
        static void Main(string[] args)
        {
            try
            {
                // Cream produse.
                ArrayList produse = new ArrayList();
                produse.Add(new Produs(1, "minge", 100.0, "de calitate"));
                produse.Add(new Produs(2, "rochie", 500.0, "cam scumpa"));
                produse.Add(new Produs(4, "joc lego", 50.0, "pentru copii"));
                Console.WriteLine("Scriem produsele:");
                foreach (Produs produs in produse)
                {
                    Console.WriteLine(produs);
                }
                Console.WriteLine();
                // cream serializatorul.
                IFormatter serializator = new BinaryFormatter();
                // Serializam produsele.
                FileStream salvam =   new FileStream("Produse.bin", FileMode.Create, FileAccess.Write);
                serializator.Serialize(salvam, produse);
                salvam.Close();
```

```
// deserializam produsele.
            FileStream incarcamFisier =   new FileStream("Produse.bin", FileMode.Open, FileAccess.Read);
            ArrayList produseSalvate = serializator.Deserialize(incarcamFisier) as ArrayList;
            incarcamFisier.Close();

            Console.WriteLine("Produse incarcate:");
            foreach (Produs produs in produseSalvate)
            {
               Console.WriteLine(produs);
            }
         }
         catch (SerializationException e)
         {
            Console.WriteLine("O execptie s-a produs la serializare!");
            Console.WriteLine(e.Message);
         }
         catch (IOException e)
         {
            Console.WriteLine("S-a produs o exceptie IO!");
            Console.WriteLine(e.ToString());
         }
         Console.ReadKey();
      }
   }
}
```

Monitorizarea fisierelor si directoarelor. De multe ori o aplicatie trebuie sa faca mai mult decat sa citeasca dintr-un fisier sau sa scrie intr-un fisier. Spre exemplu, poate fi important de stiut cand fisierele si directoarele sunt modificate.

Clasa care ne permite sa monitorizam fisierele si directoarele este FileSystemWatcher. Aceasta expune cateva evenimente pe care aplicatiile pot sa le intercepteze.

Procedura de utilizare a clasei FileSystemWatcher este urmatoarea: In primul rand trebuiesc setate o serie de proprietati care specifica unde si ce se monitorizeaza si cand trebuie lansat evenimentul pe care aplicatia urmeaza sa-l trateze. Apoi trebuie precizate adresele metodelor de tratare a evenimentelor astfel incat acestea sa fie apelate cand evenimentele sunt lansate. In final, aplicatia este pornita si se asteapta producerea evenimentelor.

Proprietatile care trebuie setate sunt:

Path	Se seteaza locatia fisierului sau directorului de monitorizat.
NotifyFilter	O combinatie de valori ale enumerarii NotifyFilters care specifica ce se urmareste in fisierele monitorizate. Acestea reprezinta proprietati ale fisierelor sau directoarelor. Daca vreo proprietate specificata se modifica atunci este lansat un eveniment. Posibilele valori ale enumerarii sunt: Attributes, CreationTime, DirectoryName, FileName, LastAccess, LastWrite, Security si Size. Acestea pot fi combinate utilizand operatorul OR.
Filter	Un filtru care specifica care fisiere se monitorizeaza (spre exemplu *.txt)

Odata setate aceste proprietati se pot trata urmatoarele evenimente: Changed, Created, Deleted si Renamed. Fiecare eveniment este lansat odata ce un fisier sau un director care satisface proprietatile: Path, NotifyFilter si Filter este modificat.

Dupa setarea evenimentelor trebuie setata proprietatea EnableRasingEvents cu true pentru a incepe monitorizarea.

Exemplu: FileSystemWatcher. La executie, includeti un parametru de comanda reprezentand directorul de monitorizat

```
using System;
using System.IO;

public class Watcher
{
    public static void Main()
    {
        Run();
    }

    public static void Run()
    {
        string[] args = System.Environment.GetCommandLineArgs();

        // If a directory is not specified, exit program.
        if (args.Length != 2)
        {
            // Display the proper way to call the program.
            Console.WriteLine("Usage: Watcher.exe (directory)");
            return;
        }
```

```
// Create a new FileSystemWatcher and set its properties.
    FileSystemWatcher watcher = new FileSystemWatcher();
    watcher.Path = args[1];
    /* Watch for changes in LastAccess and LastWrite times, and
       the renaming of files or directories. */
    watcher.NotifyFilter = NotifyFilters.LastAccess | NotifyFilters.LastWrite
       | NotifyFilters.FileName | NotifyFilters.DirectoryName;
    // Only watch text files.
    watcher.Filter = "*.txt";

    // Add event handlers.
    watcher.Changed += new FileSystemEventHandler(OnChanged);
    watcher.Created += new FileSystemEventHandler(OnChanged);
    watcher.Deleted += new FileSystemEventHandler(OnChanged);
    watcher.Renamed += new RenamedEventHandler(OnRenamed);
```

```
 // Begin watching.
        watcher.EnableRaisingEvents = true;

        // Wait for the user to quit the program.
        Console.WriteLine("Press \'q\' to quit the sample.");
        while (Console.Read() != 'q') ;
    }

    // Define the event handlers.
    private static void OnChanged(object source, FileSystemEventArgs e)
    {
        // Specify what is done when a file is changed, created, or deleted.
        Console.WriteLine("File: " + e.FullPath + " " + e.ChangeType);
    }

    private static void OnRenamed(object source, RenamedEventArgs e)
    {
        // Specify what is done when a file is renamed.
        Console.WriteLine("File: {0} renamed to {1}", e.OldFullPath, e.FullPath);
    }
}
```

Colectii, comparatii, conversii

Ne propunem sa abordam urmatoarele probleme: cum se definesc colectiile, care sunt principalele tipuri colectii, cum se compara diverse tipuri si cum se utilizeaza operatorul is, cum se definesc si cum se utilizeaza conversiile, cum se utilizeaza operatorul as.

Colectiile permit mentinerea unui grup de obiecte. Contrar tablourilor, colectiile permit realizarea unor operatii mai avansate precum controlul obiectelor pe care acestea le contin, cautarea si sortarea intr-o colectie.

Comparatii. Lucrand cu obiecte, de multe ori este necesar a le compara. Acest fapt este important atunci cand dorim sa realizam o operatie de sortare. Vom analiza modul in care putem compara doua obiecte, inclusiv supraincarcarea operatorilor, si cum se utilizeaza interfetele IComparable si IComparer.

Conversii. In multe situatii am utitlizat un cast pentru ca un obiect de un anumit tip sa fie privit ca obiect de un alt tip. Vom analiza in detaliu aceasta operatie.

Colectii. In cursurile precedente am utilizat tablourile (clasa Array din spatiul de nume System) pentru a crea tipuri de variabile care contin un numar de obiecte. Tablourile au limitele sale. Una dintre aceste limite este aceea ca tablourile au o dimensiune fixata. Nu putem adauga noi elemente intr-un tablou.

Tablourile, implementate ca instante ale clasei Array reprezinta doar unul din tipurile cunoscute drept colectii. Colectiile sunt utilizate pentru a mentine o lista de obiecte si, in general, au o functionalitate mai mare decat tablourile. O mare parte a acestei functionalitati este asigurata prin implementarea interfetelor continute de spatiul de nume System.Collections. Acest spatiu de nume contine o serie de clase care implementeaza aceste interfete.

Intrucat *functionalitatea unei colectii* (inclusiv accesarea elementelor colectiei utilizand un index) *este asigurata prin interfete*, nu suntem obligati a ne limita doar la utilizarea clasei Array. Mai degraba, putem crea propriile noastre colecti. Un avantaj este acela ca putem crea colectii puternic tipizate (strongly typed). Aceasta inseamna ca atunci cand extragem un element dintr-o colectie de acest tip nu este necesar sa utilizam un cast pentru a obtine tipul corect al elementului. Un alt avantaj este capacitatea de a expune metode specializate.

Cateva dintre intervetele spatiului de nume System.Collections, care ofera functionalitatea de baza, sunt:

IEnumerable ofera capacitatea de a cicla intr-o colectie prin intermediul metodei

GetEnumerator();

ICollection ofera capacitatea de a obtine numarul de obiecte continute de o colectie si posibilitatea de a copia elementele intr-un tablou. Mosteneste interfata IEnumerable.

IList ofera o lista a elementelor unei colectii impreuna cu capacitatea de a accesa aceste elemente. Mosteneste interfetele IEnumerable si ICollection.

IDictionary similar interfetei IList, insa ofera o lista de elemente accesibile prin intermediul unui cuvant cheie (key value) si nu un index. Mosteneste interfetele IEnumerable si ICollection.

Classa Array implementeaza primele trei interfete, insa nu suporta cateva trasaturi avansate ale interfetei IList, si reprezinta o lista cu un numar fix de elemente.

Utilizarea colectiilor. Clasa ArrayList din spatiul de nume System.Collections, la randul ei implementeaza interfetele IEnumerable, ICollection si IList, insa o face intr-un alt mod decat clasa Array. Contrar clasei Array, clasa ArrayList poate fi utilizata pentru a crea o lista cu un numar variabil de elemente.

Aplicatia de mai jos exemplifica modul in care se poate utiliza o serie de metode si proprietati implementate de ArrayList.

Exemplu: clasa ArrayList

```
using System;  using System.Collections.Generic;
using System.Collections; using System.Linq;   using System.Text;
namespace ExempluArrayList
{
    public enum TipCasa  {  Paie = 0,  Lemn = 1, Piatra = 2}
    public abstract class Animal
   {  protected string nume;
      public string Nume
      { get { return nume; }     set { nume = value; }        }
      public Animal()
      {   nume = "Animalul nu are nume“;  }
      public Animal(string numeNou)
      {   nume = numeNou;  }
      public void Hrana()
      {     Console.WriteLine("{0} a mancat.", nume);   }
   }
```

```
public class Porc : ExempluArrayList.Animal
    {
        public TipCasa Material;
        public void Casa()
        {  Console.WriteLine("{0} are o casa din {1}", nume, Material);     }

        public Porc(string numeNou, TipCasa material)
            : base(numeNou)
        {    Material = material;    }
    }

public class Gasca : ExempluArrayList.Animal
    {  public Gasca(string numeNou):base(numeNou)
        {        }
            public void FaceOua()
        {
            Console.WriteLine("{0} s-a ouat.", nume);
        }
    }
```

```
class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Creati un tablou de obiecte de tip Animal:");
            Animal[] animalArray = new Animal[2];
            Porc porculNr1 = new Porc("Nif-Nif", TipCasa.Paie);
            animalArray[0] = porculNr1;
            animalArray[1] = new Gasca("Amelia");
            foreach (Animal animalulMeu in animalArray)
            {
                Console.WriteLine("Un nou obiect de tipul {0} a fost adaugat in tablou,
        Nume = {1}",   animalulMeu.ToString(), animalulMeu.Nume);
            }
            Console.WriteLine("Tabloul contine {0} obiecte.“,   animalArray.Length);
            animalArray[0].Hrana();
            ((Gasca)animalArray[1]).FaceOua();
            Console.WriteLine();
```

```
Console.WriteLine("Creati un ArrayList de obiecte de tip Animal:");
    ArrayList animalArrayList = new ArrayList();
    Porc porculNr2 = new Porc("Nuf-Nuf", TipCasa.Lemn);
    animalArrayList.Add(porculNr2);
    animalArrayList.Add(new Porc("Naf-Naf",TipCasa.Piatra));
    animalArrayList.Add(new Gasca("Abigail"));
    foreach (Animal animalulMeu in animalArrayList)
    {
        Console.WriteLine("Un nou obiect de tipul {0} a fost adaugat in colectie,
            Nume = {1}“,  animalulMeu.ToString(), animalulMeu.Nume);
     }
    Console.WriteLine("Colectia ArrayList contine {0} obiecte.“,
                                                    animalArrayList.Count);
    ((Animal)animalArrayList[0]).Hrana();
    ((Porc)animalArrayList[1]).Casa();
    ((Gasca)animalArrayList[2]).FaceOua();
    Console.WriteLine();
```

```
        Console.WriteLine("Alte operatii in ArrayList:");
            animalArrayList.RemoveAt(1);
            ((Animal)animalArrayList[1]).Hrana();
            animalArrayList.AddRange(animalArray); //metoda virtuala

            ((Porc)animalArrayList[2]).Casa();

            Console.WriteLine("Animalul cu numele {0} are indexul {1}.",
            porculNr1.Nume, animalArrayList.IndexOf(porculNr1));

            Console.ReadKey();
        }
    }
}
```

Cum se definesc colectiile. In aplicatia precedenta, am fi putut adauga colectiei animalArrayList obiecte de orice tip. Normal ar fi ca sa putem adauga doar elemente de un anumit tip (in exemplul anterior, tipul Animal). Este momentul sa vedem cum putem defini colectii puternic tipizate (strongly typed).

O modalitate de a defini o colectie puternic tipizata este de a implementa manual metodele necesare. Insa acesta este un proces complex care necesita timp.

O alta optiune este aceea de a utiliza clasa abstracta CollectionBase din spatiul de nume System.Collections drept clasa de baza pentru colectia puternic tipizata care urmeaza a fi definita. Clasa CollectionBase expune interfetele IEnumerable, ICollection si IList, insa ofera doar o mica parte din implementarea necesara, de notat metodele Clear() si RemoveAt() ale interfetei IList si proprietatea Count a interfetei ICollection. Daca utilizatorul doreste o alta functionalitate atunci trebuie sa o implementeze el insusi.

Pentru a facilita aceasta implementare, CollectionBase ofera doua proprietati protejate care permit accesul la obiectele stocate. Se poate utiliza proprietatea List, care permite accesarea elementelor prin interfetei IList si InnerList, un obiect ArrayList utilizat pentru stocarea elementelor.

Spre exemplu, colectia care ar urma sa stocheze obiecte de tip Animal ar putea fi definita astfel:

```
public class Animalee : CollectionBase
{
  public void Add(Animal unNouAnimal)
              {  List.Add(unNouAnimal); }
  public void Remove(Animal animalulX)
              { List.Remove(animalulX);}
  public Animalee()  {  }
 }
```

Aici, Add() si Remove() au fost implementate ca metode strongly typed, prin utilizarea metodelor standard Add() si Remove() ale interfetei IList pentru a accesa elementele colectiei. Metodele expuse vor merge doar pentru clasa Animal sau clase derivate din Animal, spre deosebire de implementarea clasei ArrayList pentru care putem adauga sau extrage orice obiect.

Clasa CollectionBase faciliteaza utilizarea buclei foreach. Astfel puteti utiliza spre exemplu:

```
      Animalee colectieAnimale = new Animalee();
     colectieAnimale.Add(new Gasca(" Daffy Duck "));
     foreach (Animal animalulMeu in colectieAnimale)
  {
              Console.WriteLine ("Un nou obiect de tipul {0} a fost adaugat in colectie,
              Nume = {1}“,  animalulMeu.ToString(), animalulMeu.Nume);
  }
```

Nu este permisa accesarea elementelor colectiei prin intermediul unui index. Adica nu putem scrie:

```
colectieAnimale[0].Hrana();
```

pentru a accesa primul element din colectie, cu exceptia cazului in care colectia contine o indexare. Pentru clasa Animalee o indexare poate fi definita astfel:

```
public class Animalee : CollectionBase
{
...
public Animal this[int indexulAnimalului]
{
get { return (Animal)List[indexulAnimalului]; }
set { List[indexulAnimalului] = value; }
}
}
```

Codul indexarii de mai sus utilizeaza indexarea pusa la dispozitie de intefata IList prin intermediul proprietatii List, adica: (Animal)List[indexulAnimalului]; Castul este necesar deoarece indexarea pusa la dispozitie de IList returneaza un obiect de tipul object.

De notat ca indexarea de mai este de tipul Animal. Astfel, putem scrie

```
colectieAnimale[0].Hrana();
```

spre deosebire de cazul anterior (vezi aplicatia), unde era utilizata clasa ArrayList, care necesita utilizarea castului, spre exemplu ((Porc)animalArrayList[2]).Casa();

Drept exemplu, rescrieti aplicatia precedenta in care adaugati clasa Animalee si inlocuiti blocul metodei Main() , asa cum se specifica in urmatoarele doua slide-uri.

Adaugati clasa de mai jos:

```
public class Animalee:CollectionBase
{
    public void Add(Animal unNouAnimal)
     { List.Add(unNouAnimal);}
public void Remove(Animal animalulX)
     {List.Remove(animalulX);}
public Animalee()
{
}
public Animal this[int indexulAnimalului]
{
get
{return (Animal)List[indexulAnimalului];}
set
{List[indexulAnimalului] = value;}
}
}
```

Inlocuiti blocul metodei Main() cu cel de mai jos:

```
{       Console.WriteLine("Creati o colectie de Animalee de tip Animal:");
    Animalee animale = new Animalee();
    Porc porculNr2 = new Porc("Nuf-Nuf", TipCasa.Lemn);
    animale.Add(porculNr2);
    animale.Add(new Porc("Naf-Naf",TipCasa.Piatra));
    animale.Add(new Gasca("Abigail"));
    foreach (Animal animalulMeu in animale)
    {
        Console.WriteLine("Un nou obiect de tipul {0} a fost adaugat in colectie, Nume =
{1}",  animalulMeu.ToString(), animalulMeu.Nume);
            }
    Console.WriteLine("Colectia contine {0} obiecte.", animale.Count);
    animale[0].Hrana();
    ((Porc)animale[1]).Casa();
    ((Gasca)animale[2]).FaceOua();
    Console.WriteLine();
    Console.WriteLine("Alte operatii in ArrayList:");
    animale.RemoveAt(1);
    animale[1].Hrana();
    animale.Remove(porculNr2);
    Console.WriteLine("Au ramas {0} animale.", animale.Count.ToString());
    Console.ReadKey();
}
```

Colectii care utilizeaza cuvinte cheie. Interfata IDictionary. In locul interfetei IList, este posibil ca o colectie sa implementze in mod similar interfata IDictionary, care permite accesarea elementelor prin intermediul unor cuvinte cheie (precum un string), in locul unui index.

La fel ca in cazul colectiilor indexate, exista clasa DictionaryBase care poate fi utilizata pentru a simplifica implementarea interfetei IDictionary. Clasa DictionaryBase implementeaza, de asemenea, IEnumerable si ICollection oferind astfel facilitatile de baza pentru manipularea colectiilor.

La fel ca si CollectionBase, clasa abstracta DictionaryBase implementeaza o serie de membrii obtinuti prin intefetele suportate de aceasta clasa. Membrii Clear() si Count sunt implementati, insa RemoveAt() nu, intrucat aceasta metoda nu este continuta de interfata IDictionary.

Codul din urmatorul slide reprezinta o versiune alternativa a clasei Animalee, insa de aceasta data, este derivata din DictionaryBase.

De remarcat ca metoda Add( ) contine doi parametrii, un cuvant cheie si un obiect. Aceasta deoarece DictionaryBase contine proprietatea Dictionary care are ca tip interfata IDictionary. Aceasta interfata are metoda sa Add() care are doi parametrii (doua obiecte de tipul object), primul un cuvant cheie (a key) si cel de-al doilea un element de stocat in colectie. De asemenea, Remove() are ca parametru cuvantul cheie, aceasta inseamna ca elementul avand cuvantul cheie specificat va fi sters din colectie, iar indexarea utilizeaza un cuvant cheie de tip string si nu un index pentru a stoca elementele.

```
public class Animalee : DictionaryBase
{
public void Add(string noulID, Animal unNouAnimal)
{Dictionary.Add(noulID, unNouAnimal);}

public void Remove(string animalID)
{Dictionary.Remove(animalID);}

public Animalee()
{}
public Animal this[string animalID]
{
get { return (Animal)Dictionary[animalID];}
set { Dictionary[animalID] = value;}
}
}
```

O diferenta importanta intre colectiile bazate pe DictionaryBase si respectiv CollectionBase este faptul ca bucla foreach functioneaza oarecum diferit. Colectiile din exemplele anterioare permiteau extragerea obiectelor direct din colectie.

Utilizand foreach cu clasele derivate din DictionaryBase obtinem o structura DictionaryEntry, un alt tip definit in System.Collection. Pentru a obtine un obiect de tip Animal, trebuie sa utilizam membrul Value al acestei structuri, sau daca dorim sa obtinem cuvantul cheie utilizam membrul Key.

Astfel daca dorim sa obtinem cod echivalent cu cel de mai jos (utilizat in exemplele anterioare, unde animale reprezenta un obiect de de tipul unei clase derivate din CollectionBase):

```
foreach (Animal animalulMeu in animale)
{   Console.WriteLine("Un nou obiect de tipul {0} a fost adaugat in colectie, Nume = {1}",  animalulMeu.ToString(), animalulMeu.Nume);   }
```

atunci avem nevoie de urmatoarele instructiuni daca animale reprezenta acum un obiect de de tipul unei clase derivate din DictionaryBase):

```
foreach (DictionaryEntry myEntry in animale)
{
Console.WriteLine("Un nou obiect de tipul {0} a fost adaugat in colectie, Nume = {1}", myEntry.Value.ToString(), ((Animal) myEntry.Value).Nume);
}
```

Exercitiu: Rescrieti programul anterior folosind clasa Animalee derivata din DictionaryBase.

Comparatii. De multe ori in practica avem de *comparat doua sau mai multe obiecte*. Spre exemplu, atunci cand dorim sa *sortam sau sa cautam* intr-o colectie. Practic, realizam doua tipuri de comparatii: comparatii pentru determinarea tipurilor obiectelor *(type comparisons)* si comparatii intre valorile obiectelor *(value comparisons)*.

Type comparisons. Atunci cand dorim sa comparam doua obiecte trebuie sa cunoastem mai intai tipul obiectelor. Aceasta informatie ne permite sa determinam daca este posibila o comparatie a valorilor acestor obiecte.

O prima posibilitate de a determina tipul unui obiect este aceea de a utiliza metoda GetType(), pe care toate clasele o mostenesc de la clasa object, in combinatie cu operatorul typeof(). Spre exemplu:

```
if (obiectulMeu.GetType() = = typeof(ClasaMea))
{
        //obiectulMeu este instanta a clasei ClasaMea
}
```

Metoda GetType() returneaza tipul obiectului obiectulMeu in forma System.Type (un obiect de tipul acestei clase), in timp ce operatorul typeof() converteste numele clasei ClasaMea intr-un obiect SystemType.

De remarcat ca expresia instructiunii if se evalueaza cu true doar daca obiectulMeu este de tipul ClasaMea. Daca am considera secventa de cod de mai sus in cazul programului anterior si am lua drept obiectulMeu obiectul porculNr1 si drept ClasaMea clasa Animal (clasa de baza pentru clasa Porc) atunci expresia se evalueaza cu false.

O alta posibilitate, mult mai directa, de a determina tipul unui obiect este de a utiliza operatorul is. Acest operator permite sa determinam daca un obiect este sau poate fi convertit intr-un tip dat. In fiecare dintre aceste doua cazuri operatorul se evalueaza cu true.

Operatorul is are urmatoarea sintaxa:

<operand> is <type>

Rezultatele posibile ale acestei expresii sunt:

a) Daca <type> este o clasa atunci rezultatul este true daca <operand> este de tipul acelei clase sau de tipul unei clase derivate din clasa <type> sau poate fi impachetat in acel tip;

b) Daca <type> este o interfata atunci rezultatul este true daca <operand> este de acel tip sau daca este de un tip care implementeaza acea interfata;

c) Daca <type> este un tip valoric atunci rezultatul este true daca <operand> este de acel tip sau este de un tip care poate fi despachetat in acel tip.

De remarcat ca prima modalitate, adica cea care utilizeaza GetType(), determina doar daca un obiect este de un tip dat, in timp ce utilizand operatorul is avem mult mai multe posibilitati.

Urmatorul exemplu arata cum se poate folosi operatorul is.

```
using System;     using System.Collections.Generic;
using System.Linq;     using System.Text;
namespace Exemplu
{class Checker
{   public void Check(object param1)
{
    if (param1 is ClassA)
        Console.WriteLine("Variabila poate fi convertita in tipul ClassA.");
    else
        Console.WriteLine("Variabila NU poate fi convertita in tipul ClassA.");
    if (param1 is IInterfata)
        Console.WriteLine("Variabila poate fi convertita in IInterfata.");
    else
        Console.WriteLine("Variabila NU poate fi convertita in IInterfata.");
    if (param1 is Structura)
        Console.WriteLine("Variabila poate fi convertita in Structura.");
    else
        Console.WriteLine("Variabila NU poate fi convertita in Structura.");
}
}
```

```
interface IInterfata {}
class ClassA : IInterfata
{ }
class ClassB : IInterfata
{ }
class ClassC
{ }
class ClassD : ClassA
{ }
struct Structura : IInterfata
{ }
class Program
{   static void Main(string[] args)
  { Checker check = new Checker();
    ClassA obA = new ClassA();
    ClassB obB = new ClassB();
    ClassC obC = new ClassC();
    ClassD obD = new ClassD();
    Structura structuraS = new Structura();
    object obiect = structuraS;
Console.WriteLine("Analizam variabila
                        de tipul clasei ClassA:");
    check.Check(obA);
    Console.WriteLine("\nAnalizam variabila
                        de tipul clasei ClassB:");
    check.Check(obB);
    Console.WriteLine("\nAnalizam variabila
                        de tipul clasei ClassC:");
    check.Check(obC);
    Console.WriteLine("\nAnalizam variabila
                        de tipul clasei ClassD:");
    check.Check(obD);
    Console.WriteLine("\nAnalizam variabila
                de tipul structurii Structura:");
    check.Check(structuraS);
    Console.WriteLine("\nAnalizam variabila
     de tipul structurii Structura impachetata:");
    check.Check(obiect);
    Console.ReadKey();
}
}
}
```

Value comparisons. Sa presupunem ca avem o clasa numita Persoana care contine o proprietate de tip intreg numita Varsta. Putem compara doua obiecte de tip Persoana astfel:

```
if (persoana1.Varsta > =persoana2.Varsta)
{ ….}
```

Exista insa si alternative. Spre exemplu, in unele situatii ar fi de preferat sintaxa:

```
if (persoana1 > =persoana2)
{ ….}
```

Aceasta secventa de cod este posibila daca pentru clasa Persoana au fost supraincarcati operatorii >= si <= (a se vedea un curs anterior). Aceasta tehnica este interesanta, insa trebuie utilizata cu precautie. In codul de mai sus nu este evident ca sunt comparate varstele celor doua persoane (ar putea fi inaltimile sau masele celor doua persoane).

O alta optiune de a compara valorile obiectelor este de a utiliza interfetele IComparable si Icomparer. Aceasta tehnica este suportata de diverse clase colectii, facand-o o modalitate excelenta de sortare a obiectelor.

Interfetele IComparable si IComparer. Aceste interfete ne permit sa definim modul de comparare a obiectelor intr-un mod standard. IComparable face parte din spatiul de nume System, iar IComparer din System.Collections.

Intre cele doua interfete avem urmatoarele diferente: IComparable este implementata de clasa obiectului care urmeaza a fi comparat si permite comparatii intre acel obiect si un alt obiect; IComparer este implementata de o clasa separata si permite compararea oricaror doua obiecte.

IComparable expune metoda CompareTo(), care accepta un object ca parametru si returneza un intreg pozitiv, egal cu zero sau negativ daca obiectul instanta este mai mare, egal sau mai mic (relativ la o operatie de ordine dorita de utilizator) decat obiectul primit ca parametru. Spre exemplu, clasa Persoana ar putea implementa CompareTo() asa incat parametrul metodei sa fie de tipul clasei Persoana, iar rezultatul intors de metoda sa specifice daca persoana curenta este mai in varsta decat persoana primita ca parametru al metodei. Pentru compararea varstei s-ar putea utiliza urmatoarea secventa de cod:

```
if(persoana1.CompareTo(persoana2) = = 0)
        {Console.WriteLine(“Cele doua persoane au acceasi varsta”);}
else if(persoana1.CompareTo(persoana2) > 0)
        {Console.WriteLine(“persoana1 este mai in varsta decat persoana2”);}
else
        {Console.WriteLine(“persoana1 este mai tanara decat persoana2”);}
```

IComparer expune singura metoda Compare(), care accepta doua obiecte ca parametri si returneaza un intreg la fel ca metoda CompareTo(). Pentru un obiect, sa zicem comparaPersoane, care suporta interfata IComparer (sau altfel spus, obiectul comparaPersoane este instanta a unei clase care implementeaza interfata IComparer), se poate utiliza urmatoarea secventa de cod:

```
if(comparaPersoane.Compare(persoana1, persoana2) = = 0)
        {Console.WriteLine(“Cele doua persoane au acceasi varsta”);}
else if(comparaPersoane.Compare(persoana1, persoana2) > 0)
        {Console.WriteLine(“persoana1 este mai in varsta decat persoana2”);}
else
        {Console.WriteLine(“persoana1 este mai tanara decat persoana2”);}
```

In fiecare din cele doua cazuri, parametrii metodelor sunt de tipul object. Astfel, pentru a preveni compararea unor obiecte care nu pot fi comparate, la implementarea acestor metode, inaintea returnarii unui rezultat, trebuie sa faceti mai intai o comparare a tipurilor obiectelor si eventual lansarea unei exceptii daca obiectele nu au tipul convenit (a se vedea exemplul care sorteaza o colectie).

Arhitectura .NET include o implementare standard (default) a interfetei IComparer in cadrul clasei Comparer. Aceasta clasa face parte din System.Collections. Aceasta clasa permite comparatii specifice intre tipurile simple (inclusiv tipul string) precum si orice tip care implementeaza interfata IComparable. Spre exemplu, se poate utiliza urmatoarea secventa de cod:

```
string primulString="Primul string";
string alDoileaString= "Al doilea string";
Console.WriteLine("Daca comparam {0} cu {1} obtinem: {2}", primulString,
        alDoileaString, Comparer.Default.Compare(primulString, alDoileaString));
int primulIntreg=20;
int alDoileaIntreg= 32;
Console.WriteLine("Daca comparam {0} cu {1} obtinem: {2}", primulIntreg,
        alDoileaIntreg, Comparer.Default.Compare(primulIntreg, alDoileaIntreg));
```

Rezultatul este urmatorul:

```
Daca comparam Primul string cu Al doilea string obtinem: 1
Daca comparam 20 cu 32 obtinem: -1
```

Deoarece P este dupa A in alfabet, rezulta ca P este evaluat ca mai mare decat A si rezultatul este pozitiv. In celalalt caz, 20 este mai mic decat 32 si deci rezultatul este negativ. Rezultatul nu precizeaza magnitudinea diferentei.

A fost utilizat membrul Comparer.Default al clasei Comparer pentru a obtine o instanta a clasei Comparer. (Comparer.Default returneaza in obiect de tipul clasei Comparer).

Sortarea colectiilor utilizand interfetele IComparable si IComparer. O serie de colectii permit sortarea elementelor lor. ArrayList este un exemplu. Aceasta clasa contine metoda Sort(), care poate fi utilizata fara parametrii, caz in care sorteaza utilizand comparatorul default sau i se poate transmite ca parametru o interfata IComparer, caz in care se utilizeaza aceasta interfata pentru a compara perechi de obiecte.

Daca colectia ArrayList contine numai tipuri simple, precum intregi sau stringuri, atunci comparatorul default este foarte convenabil. Insa pentru o clasa definita de utilizator, trebuie fie implementata interfata IComparable, fie creata o alta clasa care suporta IComparer si compara obiecte de tipul primei clase. A se vedea exemplul urmator, in care se utilizeaza ambele variante ale metodei Sort().

De notat ca o serie de colectii precum CollectionBase nu expun metode pentru sortare. Daca doriti sa sortati o colectie puternic tipizata care deriva din CollectionBase atunci trebuie sa implementati manual codul pentru sortare (sau sa salvati lista colectiei intr-un ArrayList si sa utilizati apoi metoda Sort() a colectiei ArrayList).

```
using System;     using System.Collections;          using System.Collections.Generic;
using System.Linq;          using System.Text;
namespace Sortare
{
    class Persoana : IComparable
    {
       public string Nume;    public int Varsta;

       public Persoana(string nume, int varsta)
       {  Nume = nume;    Varsta = varsta;   }

       public int CompareTo(object obiect)
       {
          if (obiect is Persoana)
          {
             Persoana altaPersoana = obiect as Persoana;
             return this.Varsta - altaPersoana.Varsta;
          }
          else
          { throw new ArgumentException("Objectul cu care se compara nu este o Persoana.");   }
       }
```

```
public class ComparaNumePersoane : IComparer
    {
        public static IComparer Default = new ComparaNumePersoane();

        public int Compare(object x, object y)
        {
            if (x is Persoana && y is Persoana)
            {
                return Comparer.Default.Compare(
                  ((Persoana)x).Nume, ((Persoana)y).Nume);
            }
            else
            {
                throw new ArgumentException("Cel putin unul din obiecte nu este Persoana.");
            }
        }
    }
```

```
class Program
    {



    static void Main(string[] args)
    {
      ArrayList lista = new ArrayList();
      lista.Add(new Persoana("Ilie", 30));
      lista.Add(new Persoana("Domnica", 25));
      lista.Add(new Persoana("Simion", 27));
      lista.Add(new Persoana("Safta", 22));

      Console.WriteLine("Persoane Nesortate:");
      for (int i = 0; i < lista.Count; i++)
      {
        Console.WriteLine("{0} ({1})",
          (lista[i] as Persoana).Nume, (lista[i] as Persoana).Varsta);
      }
      Console.WriteLine();
```

```
Console.WriteLine("Persoane sortate cu comparatorul default (dupa varsta):");
            lista.Sort();
            for (int i = 0; i < lista.Count; i++)
            {
                Console.WriteLine("{0} ({1})",
                   (lista[i] as Persoana).Nume, (lista[i] as Persoana).Varsta);
            }
            Console.WriteLine();

            Console.WriteLine(
               "Persoane sortate cu comparatorul non-default (dupa nume):");
            lista.Sort(ComparaNumePersoane.Default);
            for (int i = 0; i < lista.Count; i++)
            {
                Console.WriteLine("{0} ({1})",
                   (lista[i] as Persoana).Nume, (lista[i] as Persoana).Varsta);
            }

            Console.ReadKey();
        }
    }
}
```

Conversii. De fiecare data atunci cand a trebuit sa convertim un tip intr-un altul am utilizat un cast. Se poate insa proceda si altfel.

Asa cum este posibil sa supraincarcam operatorii matematici (a se vedea unul dintre cursurile anterioare), putem defini conversii implicite si explicite intre diverse tipuri. Aceste operatii sunt utile atunci cand dorim sa convertim un tip intr-un altul, fara a fi vreo legatura intre cele doua tipuri (spre exemplu, nu exista nici o relatie de mostenire sau nu implementeaza nici o interfata comuna).

Sa presupunem ca avem definita o conversie implicita a tipului ConvClass1 in ConvClass2. Aceasta inseamna ca putem scrie

```
ConvClass1 op1 = new ConvClass1();
ConvClass2 op2 = op1;
```

Daca avem o conversie explicita atunci:

```
ConvClass1 op1 = new ConvClass1();
ConvClass2 op2 = (ConvClass2)op1;
```

Ca exemplu a modului cum pot fi definiti operatorii de conversie sa consideram codul:

```
    public class ConvClass1
{
    public int val;
    public static implicit operator ConvClass2(ConvClass1 op1)
    {
            ConvClass2 valIntoarsa = new ConvClass2();
             valIntoarsa.val = op1.val;
            return valIntoarsa;
    }
}
public class ConvClass2
{
    public double val;
    public static explicit operator ConvClass1(ConvClass2 op1)
    {
            ConvClass1 valIntoarsa  = new ConvClass1();
            checked {valIntoarsa.val = (int)op1.val;};
            return valIntoarsa;
}
}
```

Aici ConvClass1 contine o variabile de tip int, iar ConvClass2 o variabila de tip double. Deoarece o valoare intreaga poate fi convertita implicit intr-una de tip double, definim un operator implicit de conversie a clasei ConvClass1 in ConvClass2. Pentru operatia inversa, definim un operator explicit. Cu acesti operatori de conversie, au sens urmatoarele secvente de cod:

```
ConvClass 1 op1=new ConvClass1();
op1.val=3;
ConvClass2 op2=op1;
```

si respectiv (utilizand conversia explicita):

```
ConvClass 2 op4=new ConvClass2();
op4.val=3e15;
ConvClass1 op3=(ConvClass1)op4;
```

Intrucat s-a utilizat cuvantul cheie checked in definitia conversiei explicite, vom obtine o exceptie in codul precedent intrucat valoarea parametrului op4 este prea mare pentru a fi memorata de o variabila de tip int.

Operatorul as. Acest operator este utilizat pentru a converti un tip intr-un tip referinta, utilizand sintaxa: <operand> as <type>

Aceasta operatie este posibila daca operand este de tip <type>, sau daca <operand> poate fi convertit implicit in <type> sau daca <operand> poate fi impachetat in tipul <type>. Daca nu este posibila nici o conversie atunci rezultatul este null.

.NET Assembly

Introducere

Un asamblaj reprezinta un program executabil .NET (sau o parte a unui program executabil) care se prezinta ca o unitate de sine statatoare. Asamblajele reprezinta mijlocul utilizat de a impacheta programele C# pentru executie.

Atunci cand este realizat un program, *fisierul cu extensia exe obtinut in urma compilarii este un asamblaj*. Daca este contruita o biblioteca de clase (class library), *fisierul DLL (Dynamic Link Library) este deasemenea un asamblaj*. Asamblajul permite claselor, proprietatilor si metodelor publice sa fie vizibile si in alte programe. In schimb, toti membrii privati sunt mentinuti in interiorul asamblajului.

In cele ce urmeaza vom prezenta cateva aspecte legate de asamblaje. In particular, vom aborda urmatoarele:

- **O scurta trecere in revista a componentelor;**
- **Proprietatile asamblajelor, inclusiv abilitatea de a se autodescrie;**
- **Structura unui asamblaj si cum poate fi vizualizat continutul sau;**
- **Compatibilitatea versiunilor;**
- **Apelarea asamblajelor;**

Atunci cand un program C# este impachetat intr-un asamblaj, multe dintre proprietatile asamblajelor sunt destinate crearii si furnizarii unor clase speciale de programe numite componente.

O scurta trecere in revista a componentelor

O componenta este un subprogram sau o parte a unui program destinat pentru a fi utilizat de alte programe. In plus, o componenta este o unitate binara (cod executabil si nu cod sursa) care poate fi utilizata de alte programe fara a fi necesara o recompilare a codului sursa corespunzator componentei.

In sensul cel mai larg, o componenta include orice subprogram binar. Astfel orice DLL este o componenta deoarece este un subprogram continand cod executabil.

O definitie mai stricta prevede ca aceasta (componeta) sa aiba si capacitatea de a informa celelalte programe asupra continutului ei. Asamblajele au aceasta abilitate “de a-si face reclama” in .NET.

Avantajele utilizarii componentelor

Componentele ofera *posibilitatea reutilizarii subprogramelor intr-un mod flexibil*. In plus, reutilizarea unitatilor binare *salveaza timp si creste siguranta*.

Spre exemplu, sa consideram o clasa numita Shapes care contine obiecte pentru reprezentarea cercurilor, triunghiurilor sau pentru alte forme geometrice. Ar putea contine metode pentru calcularea ariei sau ar putea realiza alte operatii. Multe programe ar putea utiliza clasa Shapes: in arhitectura, inginerie, jocuri, design cu ajutorul calculatorului si altele. Astfel, nu ar fi grozav daca *routinele* pentru desenare si manipularea formelor *geometrice ar fi definite o singura data si reutilizate in toate aceste programe ?* Acesta este benficiul reutilizarii. Dar daca aceasta reutilizare ar fi *realizata fara a recompila si referi* clasa Shapes de fiecare data cand utilizam aceasta clasa ? Aceasta salveaza timp si creste siguranta deoarece elimina posibilitatea introducerii de probleme de fiecare data cand clasa este compliata si referita (compile and link). Chiar mai mult, daca o companie sau o persoana a scris aceasta componenta pe care o dorim, atunci o putem utiliza descarcand-o de pe internet sau cumparand-o, fara a fi necesar sa o scriem. *La nivel binar nu este necesar sa ne punem probleme legate de limbajul de programare utilizat pentru realizarea componentei.*

Arhitectura .NET si asamblajele furnizeaza toate aceste beneficii.

Un scurt istoric al componentelor

Pentru ca diverse programe sa utilizeze componente la nivel binar, trebuie sa existe ceva standard pentru implementarea modului cum sunt numite si utilizate clasele si obiectele la nivel binar. Metoda standard care realizeaza acest lucru in ceea ce priveste produsele Microsoft a evoluat de-a lungul timpului.

Microsoft Windows a introdus DLL (Dynamic Link Library) unde unul sau mai multe programe pot utiliza o parte din codul stocat intr-un fisier separat. Aceasta a functionat la nivelul de jos daca programele erau scrise in acelasi limbaj (uzual C). Totusi, programele trebuiau sa cunoasca in avans o multime de informatii despre DLL-ul pe care il utilizau si DLL-urile nu permiteau ca programele sa isi poata schimba date intre ele.

Pentru a permite schimbul de date, a fost creat DDE (Dynamic Data Exchange). Acesta a definit un format si un mecanism pentru a trimite date dintr-un program in altul, insa nu a fost suficient de flexibil. A urmat OLE 1.0 (Object Linking and Embedding) care a permis unui document precum fisierul Word sa contina un document dintr-un alt program (precum Excel). OLE 1.0 functiona ca o componenta, insa nu era o componenta standard.

Microsoft a definit prima sa componenta standard odata cu COM (Component Object Model), la mijlocul anilor 90. OLE 2.0 si alte tehnologii succesoare au fost construite pe baza lui COM. Distributed COM (DCOM) a introdus capacitatea componentelor COM de a interactiona in retea. COM functioneaza bine, insa este dificil de invatat (mai ales cand se utilizeaza din C++) si utilizat. COM necesita informatii despre componente, pentru inserarea lor in registri, facand instalarea complexa iar dezinstalarea dificila. COM a fost initial destinata utilizarii cu C si C++, fiind apoi extinsa la Visual Basic. De-a lungul timpului au fost semnalate mai multe probleme legate de DLL-uri. Intrucat utilizatorii puteau instala versiuni multiple ale DLL-urilor si componentelor COM furnizate de Microsoft sau alte companii, era foarte usor ca un program sa instaleze o versiune diferita a unui DLL deja utilizat de un alt program iar aceasta putea sa cauzeze caderea programului initial. Povara urmaririi tuturor informatiilor despre diferite DLL-uri instalate pe un system a facut dificila upgradarea si mentinerea componentelor.

Programarea .NET aduce un nou standard care se adreaseaza acestor probleme, asamblajul .NET (.NET assembly).

Proprietatile asamblajelor, inclusiv abilitatea de a se autodescrie

Inainte de a analiza structura unui asamblaj, sa discutam mai intai cateva proprietati ale asamblajelor .NET.

Autodescrierea

Cel mai important aspect al asamblajelor .NET, care le diferentiaza fata de predecesoare, este acela de a se autodescrie. Descrierea este continuta in asamblaj si astfel sistemul sau programul care apeleaza asamblajul nu trebuie sa caute informatii in registri sau in alta parte despre obiectele continute intr-un asamblaj.

Autodescrierea asamblajelor .NET trece dincolo de numele obiectelor, metodelor sau tipul de date al parametrilor. Un asamblaj .NET contine informatii *despre versiunea obiectelor* (spre exemplu Shapes 1.0 urmat de Shapes 1.1 sau Shape 2.0) *si controleaza securitatea obiectelor* continute. Toate aceste informatii sunt continute in asamblaj si nu este necesar ca programele care utilizeaza acest asamblaj sa caute informatii in alta parte. Aceasta face instalarea unei componente .NET mult mai usoara si mai directa decat tehnologiile Windows precedente. In fapt, trebuie literalmente copiat asamblajul pe discul dorit.

Asamblajele .NET si biblioteca de clase .NET

Fiecare program .NET, inclusiv programele C#, utilizeaza biblioteca de clase .NET. Aceste clase sunt apelate de fiecare data cand este apelata o metoda dintr-un spatiu de nume (spre exemplu spatiul de nume System). Fiecare clasa a acestei biblioteci este parte a propriului sau asamblaj. Spre exemplu, clasele care deseneaza sunt continute in asamblajul System.Drawing.dll. Daca atunci cand editati un program adaugati o referinta catre System.Drawing.dll, compilatorul va include o referinta catre acest asamblaj cand este construit asamblajul programului dumneavoastra. *La executie, CLR (Common Language Runtime) citeste metadatele din asamblajul programului dumneavoastra pentru a determina care sunt celelalte asamblaje necesare, iar apoi le localizeaza si le incarca pentru ca programul sa le utilizeze. Asamblajele programului dumneavoastra pot sa refere alte asamblaje. Proprietatea asamblejelor de a se autodescrie face posibila urmarirea tuturor referinelor fara a fi necesar ca programatorul sa le cunoasca*.

Corespondenta dintre spatiile de nume si asamblaje nu este bijectiva. Un asamblaj poate contine informatii din mai multe spatii de nume si reciproc un spatiu de nume poate fi impartit in mai multe asamblaje.

Programarea in limbaj mixt

Un beneficiu al asamblajelor .NET il reprezinta *programarea in limbaj mixt*, deoarece componentele pot fi apelate din oricare din limbajele .NET (C++, C#, Visual Basic) neavand importanta in care din limbaje au fost scrise acele componente.

Pentru a permite programarea in limbaj mixt, .NET asigura urmatoarele:

- CLR (Common Language Runtime) care gestioneaza executia tuturor asamblajelor .NET necesare programului;

-MSIL (Microsoft Intermediate Language) este un pseudocod sau limbaj intermediar generat de compilatoarele limbajelor .NET. Acest pseudocod standard este executat de CLR. De asemenea, CLR defineste formatul pentru pastrarea metadatelor asamblajelor. Aceasta inseamna ca asamblajele, oricare ar fi limbajul in care au fost scrise, au un format comun pentru a pastra metadatele;

- Common Language Specification (CLS) defineste trasaturile pe care limbajele trebuie sa le indeplineasca pentru a permite interoperabilitatea cu alte limbaje .NET.

- Common Type System (CTS) defineste tipurile de baza ale tuturor limbajelor .NET si regulile pentru definirea propriilor noastre clase.

In baza specificatiilor CLS rezulta ca putem scrie o componenta in C#, iar asamblajul care contine componenta poate fi utilizat de un alt program scris intr-un alt limbaj .NET precum Visual Basic .NET deoarece atat C# cat si Visual Basic .NET vor fi executate de CLR. Similar, programele C# pot utiliza componente scrise in Visual C++ .NET etc. La nivel de asamblaj, toate clasele, obiectele, tipurile de date utilizate de limbajele .NET sunt puse in comun, incat utilizatorul poate mosteni clase si utiliza componente indiferent de limbajul in care acestea au fost scrise.

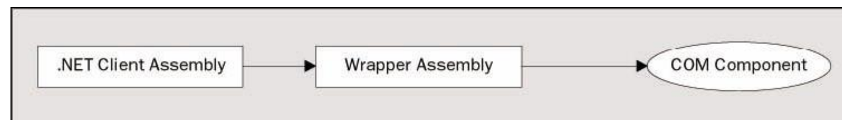
Interoperabilitatea cu COM sau cu alte tehnologii

Arhitectura .NET permite componentelor sau bibliotecilor scrise utilizand COM sau alte tehnologii sa fie utilizate cu C# sau alte limbaje .NET.

Acest mecanism functioneaza via proprietatii de autodescriere a asamblajului; Este creat un asamblaj (wrapper assembly) care “infasoara” componentele COM sau ale altor tehnologii in asa fel incat acestea sa se poata autodescrie pentru a fi executate in .NET. Asamblajul wrapper converteste tipurile de date COM la tipurile .NET si permite schimbul de apeluri de la limbajele .NET la COM si viceversa.

Visual Studio .NET creaza automat asamblajul wrapper atunci cand se adauga o referinta la o componenta COM (a se vedea dialogul Add Reference).

Diagrama de mai jos arata modul de lucru al asamblajului wrapper. Apelurile facute de asamblajul .NET client trec prin intermediul asamblajului wrapper pentru a ajunge la componenta COM. Din punct de vedere al asamblajului .NET, componenta este asamblajul wrapper si nu componenta COM.



Structura unui asamblaj si cum poate fi vizualizat continutul sau

Partile unui asamblaj mijlocesc programelor .NET aflarea existentei unora despre altele si rezolvarea problemei referintelor dintre programe si componente.

Asamblajele .NET formate dintr-un singur fisier au formatul din diagrama alaturata.

Un asamblaj contine:

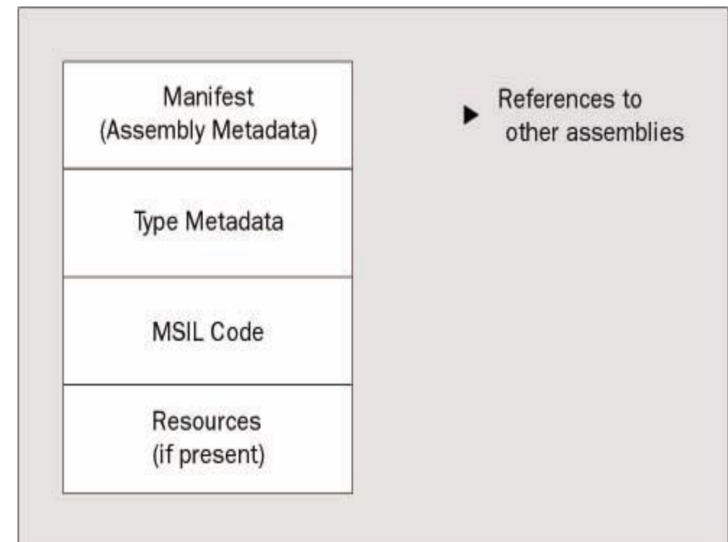
a) **metadatele** (date care descriu alte date) care permit altor programe sa caute clase, metode si proprietati ale obiectelor definite in asamblaj (**Manifest** si **Type Metadata**);

b) codul executabil al unui program sau al unei biblioteci de clase (**MSIL Code**);

c) resursele (daca exista). Resursele sunt partile neexecutabile ale unui program (precum imagini, iconite sau mesaje).

Fiecare asamblaj contine un **manifest** care descrie continutul asamblajului. El contine in esenta trei tipuri de informatii: informatii despre asamblajele externe pe care le refera, informatii despre asamblajul propriu-zis si modulele pe care le contine. El sta la baza conceptului de autodefinire.

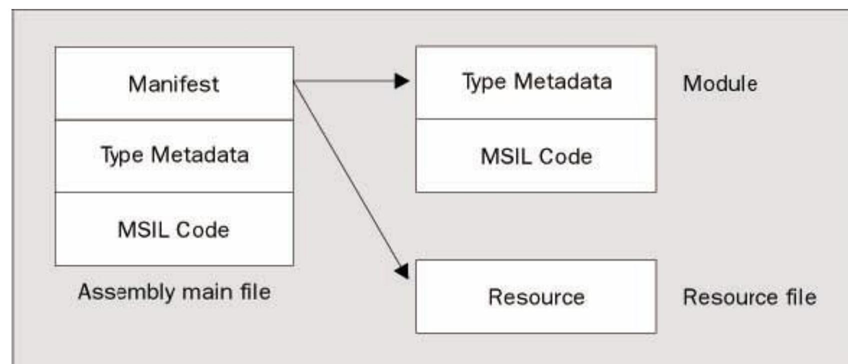
Manifestul este urmat de **metadata type**, o descriere a claselor, proprietatilor, metodelor etc. continute de asamblaj, impreuna cu tipurile de date si valorile returnate. Apoi urmeaza **codul binar** pentru fiecare tip stocat in format MSIL.



Desi un asamblaj consta de cele mai multe ori dintr-un singur fisier dll sau exe, este posibil ca acesta sa contina mai multe fisiere (vezi diagrama).

Din punct de vedere al .NET-ului un asamblaj format din mai multe fisiere este privit ca o singura unitate logica care se intampla sa consiste din mai multe fisiere. Doar unul dintre fisiere contine manifestul. Manifestul indica celelalte fisiere ale asamblajului. Fisierele care contin cod executabil se numesc module. Modulele sunt asadar formate din tipuri metadata (metadata types) si cod MSIL. Ar putea avea si fisiere resurse care nu contin cod executabil.

Asamblajele formate din mai multe fisiere sunt utilizate doar in unele aplicatii avansate. Un modul este incarcat doar atunci cand este executat sau utilizat. Aceasta pentru a salva timp si memorie daca modulul nu este utilizat frecvent.



Exemplu: Vom crea un proiect de tip Class Library numit Shapes avand codul de mai jos.

```
namespace Shapes{
public class Circle{
double Radius;
public Circle(){
Radius = 0;
}
public Circle(double givenRadius){
Radius = givenRadius;
}
public double Area(){
return System.Math.PI * (Radius * Radius);
}
}
public class Triangle{
double Base;
double Height;
public Triangle(){
Base = 0;
Height = 0;
}
public Triangle(double givenBase, double givenHeight){
Base = givenBase;
Height = givenHeight;
}
public double Area(){
return 0.5F * Base * Height; // area = 1/2 base * height
}
}
}
```

Odata construit asamblajul (Shapes.dll), vom utiliza Ildasm (**I**ntermediate **L**anguage **D**is**AS**se**M**bler) pentru a-i vedea continutul.

Adaugam apoi o metoda, spre exemplu

```
using System Drawing;
public void Draw()
{
Pen p = new Pen(Color.Red);
}
```

Pentru ca proiectul sa poata fi compilat trebuie adaugata o referinta catre System.Drawing.dll (in Solution Explorer\ References, click dreapta si Add References).

Se observa ca pe langa asamblajele externe, manifestul contine o multime de informatii despre el insusi. Acestea se numesc atribute ale asamblajului. Se observa ca in Solution Explorer\ Properties se afla un fisier numit AssemblyInfo.cs. Acest fisier descrie atributele asamblajului, inclusiv versiunile sale. Cu Ildasm se pot vedea valorile acestor atribute.

Unul dintre atribute il reprezinta AssemblyVersion attribute. Versiunea unui asamblaj .NET are patru parti. Valorile atributului au urmatoarea semificatie: primul numar reprezinta Major Revision, al doilea Minor Revision, apoi Build Number si Revision. Primele doua numere prezinta o mare importanta in determinarea compatibilitatii versiunilor. Ultimele doua valori pot fi setate prin default de catre Visual Studio.

Compatibilitatea versiunilor

La executia unui program, .NET verifica daca versiunile sunt compatibile.

Asa cum se poate observa, manifestul unui asamblaj contine numarul versiunii asamblajului curent si numerele versiunii asamblajelor externe referite. Atunci cand .NET incarca un asamblaj extern, ii verifica manifestul pentru a compara vesiunile. Daca asamblajele au numerele Major sau Minor diferite atunci versiunile sunt incompatibile si asamblajul referit nu va fi incarcat.

Spre exemplu Shapes 1.1 nu este compatibil cu Shapes 2.0 2.1, 1.0 sau 1.2.

Ce se intampla atunci cand sistemul contine un program A care utilizeaza Shapes 1.0 si un program B care utilizeaza Shapes 1.1? De acest lucru se ocupa .NET prin intermediul unei caracteristici intitulate “executie diferentiata” (side by side execution) care face ca atat Shapes 1.0 cat si Shapes 1.1 sa fie ambele instalate pe acelasi calculator si sa fie disponibile programelor care necesita versiunile corespunzatoare.

Doua asamblaje avand aceleasi numere Major si Minor dar avand numarul Build diferit pot fi sau nu compatibile. La rulare, se presupune ca sunt compatibile astfel ca se permite ca acestea sa fie incarcate. Depine insa de dezvoltator daca modificarile aduse nu pericliteaza executia acestuia.

Apelarea asamblajelor

Sa presupunem ca asamblajul Shapes.dll este apelat de de programul de mai jos:

```
using System;
using Shapes;
namespace ShapeUser{
public class ShapeUser{
public static void Main(){
Circle c = new Circle(1.0F);
Console.WriteLine("Area of Circle(1.0) is {0}", c.Area());
}
}
}
```

Pentru ca programul sa poata fi compilat trebuie adaugata o referinta la asamblajul Shapes.dll Acest lucru se realizeaza din Solution Explorer, se selecteaza References, iar apoi prin click dreapta se deschide fereastra Add References, de unde se alege asamblajul dorit. Astfel, o copie a asamblajului este adaugata in directorul Debug. Programul poate acum fi construit, iar continutul asamblajului obinut poate fi analizat cu Ildasm.

Introducere in Windows Forms

Multe dintre sistemele de operare creaza evenimente si utilizeaza ferestere pentru a interactiona cu utilizatorii. In ceea ce priveste Microsoft Windows, pentru a crea o fereastra sunt utilizate rutinele bibliotecii Win32.

Arhitectura .NET contine un numar impresionant de clase, structuri, interfete, metode etc. (Base Class Libraries) utile pentru crearea de aplicatii, iar unele dintre acestea sunt destinate pentru a crea ferestre. Ceea ce in programarea Windows se numeste o fereastra, in programarea .NET se numeste form. Acest form este caracterizat de un titlu, contine bare de navigare, un meniu, toolbars (toate acestea sunt optionale) iar ceea ce ramane se numeste zona client.

Pentru a crea un form creati in fapt un obiect de tipul unei clase care mosteneste clasa Form din spatiul de nume System.Windows.Forms. Clasa Form este derivata din ContainerControl care este derivata la randul ei din ScrollableControl. In fapt, lantul mostenirilor este:

Object-MarshalByRefObject-Component-Control-ScrollableControl-ContainerControl-Form

Form-urile sunt obiecte care expun proprietati ce definesc modul de afisare si metode care raspund evenimentelor ce definesc interactiunea cu utilizatorul. Setand proprietatile unui form si scriind codul ce raspunde la evenimente, in fapt realizam obiecte care se comporta conform cerintelor cerute de aplicatie. Form-urile sunt instante ale clasei Form (sau ale unor clase derivate din aceasta).

Cele mai simple exemple

In primele doua exemple se creaza obiecte instanta (forms) ale clasei MyForm si se utilizeaza metoda Show( ) si respectiv proprietatea Visible pentru a face vizibile ferestrele (form-urile) create. Atunci cand este rulat unul din primele doua exemple, programul arata form-ul si avanseaza la linia urmatoare, care este de fapt sfarsitul programului. Deoarece programul a ajuns la final, formul se inchide. Astfel, form-ul se observa doar pentru o fractiune de secunda.

Pentru a crea un form care sa ramana pe ecran se utilizeaza metoda Run() din clasa Application. Clasa Application face parte din acelasi spatiu de nume System.Windows.Forms, este derivata direct din clasa Object si toate metodele sale sunt statice.

Spre deosebire de metoda Show() care dupa ce arata form-ul si avanseaza la linia urmatoare, metoda Run() creaza o bucla (loop) care face ca programul sa ruleze in continuare pana cand este produs un eveniment care incheie bucla (de regula acest eveniment se produce prin apasarea unui buton Cancel).

Exemplul 1:

```
using System.Windows.Forms;
class MyForm:Form
{public static void Main()
{MyForm myfirstform=new MyForm( );
myfirstform.Show( );} }
```

Exemplul 2:

```
using System.Windows.Forms;
class MyForm:Form
{public static void Main()
{MyForm myfirstform=new MyForm( );
myfirstform.Visible=true;} }
```

Exemplul 3:

```
using System.Windows.Forms;
class MyForm:Form
{public static void Main(string [] args)
{MyForm myfirstform=new MyForm( );
myfirstform.Text="";
if (args.Length>0)
{ for (int i=0;i<args.Length; i++)
myfirstform.Text+=args[i]+" ";}
Application.Run(myfirstform);}
}
```

Crearea unui form

Clasa Form contine un numar mare de metode, proprietati, evenimente pentru a putea fi descrise intr-o carte. Functionalitatea acestora poate fi consultata prin documentatia online. Indicii privind functionalitatea membrilor clasei Form pot fi gasite si in mediul de programare Visual Studio .NET.

Este util sa studiem cateva dintre acestea. Vom incepe prin a descrie cateva proprietati asociate barei de titlu.

Bara de titlu (control box) ofera utilizatorului informatii despre program. In exemplele precedente s-a observat ca aceasta contine cativa itemi precum Minimize, Maximize, si Close. Acestia pot fi controlati prin intermediul proprietatilor:

ControlBox (determina daca control box este afisat), HelpButton (indica daca butonul de help este afisat. Acest buton este disponibil doar daca MinimizeBox si MaximizeBox sunt ambii false) MaximizeBox (indica daca butonul Maximize este inclus) MinimizeBox (indica daca butonul Minimize este inclus), Text (include titlul ferestrei).

Exemplul 4

```
using System.Windows.Forms;
public class FormApp : Form
{
    public static void Main( string[] args )
    {
    FormApp frmHello = new FormApp();
    // Caption bar properties
    frmHello.MinimizeBox = true;
    frmHello.MaximizeBox = false;
    frmHello.HelpButton = true;
    frmHello.ControlBox = true;
    frmHello.Text = @"My Form’s Caption";

    Application.Run(frmHello);
    }
 }
```

Dimensionarea unui form

Urmatorul lucru care trebuie controlat reprezinta dimensiunea form-ului. Se pot utiliza o serie larga de metode si proprietati care realizeaza acest lucru. Exemple de proprietati care gestioneaza dimensiunile unui form:

AutoSize The form automatically adjusts itself, based on the font or controls used on it.

AutoSizeMode The base size used for autoscaling the form.

AutoScroll The form has the automatic capability of scrolling.

AutoScrollMargin The size of the margin for the autoscroll.

AutoScrollMinSize The minimum size of the autoscroll.

AutoScrollPosition The location of the autoscroll position.

ClientSize The size of the client area of the form.

DefaultSize The protected property that sets the default size of the form.

DesktopBounds The size and location of the form.

DesktopLocation The location of the form.

Height The height of the form

MaximizeSize The maximum size for the form.

MinimizeSize The minimum size for the form.

Size The size of the form. set or get a Size object that contains an x, y value.

SizeGripStyle The style of the size grip used on the form.
A value from the SizeGripStyle enumerator. Values are Auto (automatically displayed when needed), Hide (hidden), or Show (always shown).

StartPosition The starting position of the form. This is a value from the FormStartPosition enumerator. Possible FormStartPosition enumeration values are CenterParent (centered within the parent form), CenterScreen (centered in the current display screen), Manual (location and size determined by starting position), WindowsDefaultBounds positioned at the default location), and WindowsDefaultLocation (positioned at the default location, with dimensions based on specified values for the size).

Width The width of the form.

Exemplul 5

```
using System.Windows.Forms;

public class FormSize : Form
{
public static void Main( string[] args )
 {
 FormSize myForm = new FormSize();
 myForm.Text = "Form Sizing";

 myForm.Width = 400;
 myForm.Height = 100;

  myForm.StartPosition =
       FormStartPosition.Manual;


 Application.Run(myForm);
 }
 }
```

Schimbarea culorilor si schimbarea fundalului unui form

Pentru a schimba culoarea de fundal a unui form se utilizeaza proprietatea BackColor. Culorile pot fi luate din structura Color aflata in spatul de nume System.Drawing.

Pentru a schimba culoarea de fundal se poate utiliza o imagine de fundal importata dintr-un fisier. O imagine este setata prin intermediul proprietatii BackgroundImage care pastreaza ca valoare un obiect de tip Image.

Clasa Image din spatiul de nume System.Drawing contine metoda statica FromFile( ) care returneaza un obiect de tip Image. Un obiect din clasa Image include proprietatile Width si Height. In program inaltimea si lungimea zonei client sunt setate in asa fel incat sa aiba aceeasi valoare ca si inaltimea si latimea imaginii.

Exemplul 6

```
using System.Windows.Forms;
using System.Drawing;

public class PicForm : Form
{
public static void Main( string[] args )
 {
 PicForm myForm = new PicForm();
 myForm.BackColor = Color.HotPink;
 myForm.Text = "PicForm - Backgrounds";

 if (args.Length >= 1)
 {
 myForm.BackgroundImage = Image.FromFile(args[0]);

 Size tmpSize = new Size();
 tmpSize.Width = myForm.BackgroundImage.Width;
 tmpSize.Height = myForm.BackgroundImage.Height;
 myForm.ClientSize = tmpSize;

 myForm.Text = "PicForm - " + args[0];
 }

 Application.Run(myForm);
 }
 }
```

Schimbarea frontierei (border) unui form

Controlarea border-ul unui form este necesara din punct de vedere estetic. De asemenea determina daca un form poate fi redimensionat sau nu. Pentru a modifica un border trebuie setata proprietatea FormBorderStyle cu una din urmatoarele enumeratii:

Fixed3D The form is fixed (not resizable) and has a 3D border.

FixedDialog The form is fixed (not resizable) and has a thick border.

FixedSingle The form is fixed (not resizable) and has a single-line border.

FixedToolWindow The form is fixed (not resizable) and has a tool window border.

None The form has no border.

Sizeable The form is resizable.

SizeableToolWindow The form has a resizable tool window border.

Exemplul 7

```
using System.Windows.Forms;
using System.Drawing;

public class BorderForm : Form
{
 public static void Main( string[] args )
 {
 BorderForm myForm = new BorderForm();
 myForm.BackColor = Color.SteelBlue;
 myForm.Text = "Borders";

 myForm.FormBorderStyle =
      FormBorderStyle.Fixed3D;

 Application.Run(myForm);
 }
 }
```

Adaugarea componentelor si control-erelor

Cuvantul *control* este folosit pentru a se referi *la obiectele de interfata* cum ar fi *butoane, bare de defilare (scroll bars), campuri de editare (labels),* etc.

Clasa Control implementeaza aspectul vizual al Form-urilor, iar fiecare obiect particular (buton, campuri de editare, etc.) implementeaza particularitati specifice si stie sa raspunda la mesajele adresate direct sau indirect.

Tehnic termenul control refera orice obiect .NET care implementeaza clasa Control. In practica, utilizam cuvantul control pentru a ne referi la orice obiect vizual gazduit de un form. Aceasta este in contrast cu o componenta care are la fel de multe caracteristici asemanatoare cu un control insa nu expune o interfata vizuala. Spre exemplu un *button* este un control, insa un *timer* este o componenta.

In mediul de programare Visual Studio.Net, control-erele, componentele cat si containerele (acestea din urma grupeaza un set de control-ere) se gasesc in Toolbox.

In exemplul urmator sunt implementate un label si un button. De asemenea, notati modul in care putem raspunde cu un anumit cod la producerea unui eveniment.

Exemplul 8 (implementarea unui buton si a unui label, 2 pagini) pagina 1

```
using System;
using System.Windows.Forms;
using System.Drawing;

public class ButtonApp : Form
{
private Label myDateLabel;
private Button btnUpdate;

public ButtonApp()
{ InitializeComponent();}

//INSEREAZA METODA InitializeComponent() AICI

protected void btnUpdate_Click( object sender, System.EventArgs e)
{ DateTime currDate =DateTime.Now ;
this.myDateLabel.Text = currDate.ToString(); }

protected void btnUpdate_MouseEnter(object sender, System.EventArgs e)
{ this.BackColor = Color.HotPink;}

protected void btnUpdate_MouseLeave(object sender, System.EventArgs e)
{ this.BackColor = Color.Blue;}

protected void myDataLabel_MouseEnter(object sender, System.EventArgs e)
{ this.BackColor = Color.Yellow;}

protected void myDataLabel_MouseLeave(object sender, System.EventArgs e)
{ this.BackColor = Color.Green; }

public static void Main( string[] args )
{ Application.Run( new ButtonApp() );   /* creaza fereastra*/      }
}
```

```
private void InitializeComponent()
 { this.Text = Environment.CommandLine;
 this.StartPosition = FormStartPosition.CenterScreen;
 this.FormBorderStyle = FormBorderStyle.Fixed3D;

 myDateLabel = new Label(); // Creaza label

 DateTime currDate ;
 currDate = DateTime.Now;
 myDateLabel.Text = currDate.ToString();

 myDateLabel.AutoSize = true;
 myDateLabel.Location = new Point( 50, 20);
 myDateLabel.BackColor = this.BackColor;

 this.Controls.Add(myDateLabel); // Adauga label-ul ferestrei

 this.Width = (myDateLabel.PreferredWidth + 100); // Seteaza latimea ferestrei pe baza latimii labelui

 btnUpdate = new Button(); // Creaza button

 btnUpdate.Text = "Update";
 btnUpdate.BackColor = Color.LightGray;
 btnUpdate.Location = new Point(((this.Width/2) -
(btnUpdate.Width / 2)), (this.Height - 75));

 this.Controls.Add(btnUpdate); // Adauga button-ul ferestrei

 btnUpdate.Click += new System.EventHandler(this.btnUpdate_Click);
 btnUpdate.MouseEnter += new System.EventHandler(this.btnUpdate_MouseEnter);
 btnUpdate.MouseLeave += new System.EventHandler(this.btnUpdate_MouseLeave);
 myDateLabel.MouseEnter += new System.EventHandler(this.myDataLabel_MouseEnter);
 myDateLabel.MouseLeave += new System.EventHandler(this.myDataLabel_MouseLeave);
 }
```

Crearea unui form utilizand Visual Studio .NET

Atunci cand creati un form ar trebui sa incercati sa utilizati urmatoarele recomandari:

1. Aliniati labelurile la dreapta form-ului si utilizati doua puncte (:) dupa text;
2. Alegeti numele controlerelor asa incat acestea sa descrie scopul lor. Utilizati un prefix pentru a indica tipul controlerului;
3. Utilizati culorile default si fonturile default;
4. Utilizati controlerul corespunzator interactivitatii pe care o doriti;
5. Aliniati butoanele OK si Cancel in coltul din dreapta jos al ferestrei;
6. Mentineti controlerele si labelurile aliniate in coloane imaginare;
7. Utilizati tabindexul pentru a naviga prin intermediul tastaturii.

Introducere in Windows Forms
Control-ere

Control-ere

Control-erele din .NET deriva din clasa System.Windows.Forms.Control Acesta clasa defineste functionalitatea control-erelor. Din acest motiv, o mare parte a proprietatilor si evenimentelor control-erelor sunt identice.

O parte a claselor derivate din clasa Control sunt la randul lor clase de baza pentru alte controlere, cum este spre exemplu cazul claselor Label si TextBoxBase, din diagrama de mai jos:

```
Object --- MarshalByRefObject --- Component --- Control --- Label --- LinkLabel
                                                        --- TextBoxBase --- TextBox
                                                                        --- RichTextBox
                                                        --- ButtonBase --- Button
                                                                       --- CheckBox
                                                                       --- RadioButton
                                                        --- ListView
                                                        --- etc.
```

Unele controlere (numite user controls) deriva din clasa *System.Windows.Forms.UserControl* care deriva la randul ei din clasa Control si asigura functionalitatea necesara pentru a ne crea propriile noastre controlere. Controlerele utilizator utilizate pentru a crea interfete Web design deriva dintr-o alta clasa, si anume *System.Web.UI.Control.*

Clasa Control contine un numar mare de proprietati, evenimente si metode.

Proprietati ale clasei Control

Toate controlerele au nu numar mare de proprietati care sunt utilizate pentru a manipula comportarea unui anumit controler. Clasa de baza a controlerelor, System.Windows.Forms.Control, are un numar de proprietati pe care celelalte controlere le mostenesc direct, sau le redefinesc pentru a oferi comportari specifice.

Mai jos sunt trecute in revista cateva dintre proprietatile clasei Control. Pentru o descriere detaliata a tuturor proprietatilor, este indicat sa consultam documentatia .NET Framework SDK.

Nume	Descriere
Anchor	Gets or sets the edges of the container to which a control is bound and determines how a control is resized with its parent.
BackColor	Gets or sets the background color for the control.
BackgroundImage	Gets or sets the background image displayed in the control.
Dock	Gets or sets which control borders are docked to its parent control and determines how a control is resized with its parent.
Enabled	Gets or sets a value indicating whether the control can respond to user interaction.
ForeColor	Gets or sets the foreground (prim-plan) color of the control.
Name	Gets or sets the name of the control.This name can be used to reference the control in code.
TabIndex	Gets or sets the tab order of the control within its container.
TabStop	Gets or sets a value indicating whether the user can give the focus to this control using the TAB key.
Tag	Gets or sets the object that contains data about the control.This value is usually not used by the control itself and is there for you to store information about the control on the control itself. When this property is assigned a value through the Windows Form designer, you can only assign a string to it.
Text	Gets or sets the text associated with this control.
Visible	Gets or sets a value indicating whether the control and all its parent controls are displayed.

Evenimente ale clasei Control

Evenimentele generate de controlerele din Windows Forms sunt de regula asociate actiunii utilizatorului. Spre exemplu, atunci cand utilizatorul apasa cu mouse-ul pe un button, acel button genereaza un eveniment care specifica ce i s-a intamplat. Tratarea evenimentului este modalitatea prin care programatorul ofera functionalitate button-ului respectiv.

Clasa Control defineste un numar mare de evenimente comune controlerelor. Mai jos sunt trecute in revista cateva dintre evenimentele clasei Control. Pentru o descriere detaliata a tuturor evenimentelor, este indicat sa consultam documentatia .NET Framework SDK.

Nume	Descriere
Click	Occurs when the control is clicked.
DoubleClick	Occurs when the control is double-clicked. Handling the Click event on some controls, such as the Button control will mean that the DoubleClick event can never be called.
KeyDown	Occurs when a key becomes pressed while the control has focus. This event always occurs before KeyPress and KeyUp.
KeyPress	Occurs when a key becomes pressed while a control has focus. This event always occurs after KeyDown and before KeyUp. The difference between KeyDown and KeyPress is that KeyDown passes the keyboard code of the key that has been pressed, while KeyPress passes the corresponding char value for the key.
KeyUp	Occurs when a key is released while a control has focus. This event always occurs after KeyDown and KeyPress.
Validated	This event is fired when a control with the CausesValidation property set to true is about to receive focus. It fires after the Validating event finishes and indicates that validation is complete.
Validating	Fires when a control with the CausesValidation property set to true is about to receive focus. Note that the control which is to be validated is the control which is losing focus, not the one that is receiving it.

Tratarea unui eveniment

Exista trei posibilitati pentru a insera codul de tratare pentru un anumit eveniment. Primul dintre acestea incepe prin a face dublu-click pe control-erul avut in discutie. Suntem dusi la metoda care trateaza evenimentul asociat prin default control-erului. (Daca controlerul este un Button evenimentul asociat prin default este Click, daca este un TextBox atunci evenimentul este TextChanged etc.) Daca acesta este evenimentul dorit atunci codul dorit se introduce in blocul metodei care trateaza evenimentul.

Daca insa se doreste un un alt eveniment decat evenimntul default atunci sunt doua posibilitati. Una dintre acestea este de a alege evenimentul dorit din lista de evenimente a ferestrei Properties. Evenimentul scris pe fond gri este evenimentul default. Pentru a alege un alt eveniment si a adauga metoda de tratare a evenimentului se face dublu click pe evenimentul dorit din lista de evenimente. Metoda care trateaza evenimentul considerat este generata automat, numele metodei fiind foarte sugestiv. Spre exemplu, daca controlerul se numeste textBox1, evenimentul considerat este Click atunci metoda care trateaza evenimentul se va numi textBox1_Click. Altfel, puteti tasta numele dorit de dumneavoastra pentru metoda de tratare a evenimentului in dreptul evenimentului dorit, iar apoi sa apasati enter. Metoda avand numele dorit de dumneavoastra va fi generata automat.

O alta optiune este de a adauga codul care creaza evenimentul in blocul constructorului, dupa apelul metodei InitializeComponent(), iar apoi de a adauga metoda care trateaza evenimentul, bineinteles in afara blocului constructorului.

Aceste ultime doua posibilitati necesita doi pasi: crearea evenimentului si crearea metodei de tratare a evenimetului.

In cele ce urmeaza vom prezenta utilitatea catorva control-erele (Clase derivate din clasa Control a caror instante asigura o functionaltate specifica).

Clasa ButtonBase si clasele sale derivate

Atunci cand ne gandim la un Button, ne imaginam probabil un buton dreptunghiular care realizeaza anumite sarcini atunci cand facem click pe acesta. Desi aceasta este principala menire a unui button, totusi, .NET ofera o clasa derivata din clasa Control, si anume System.Windows.Foms.ButtonBase care implementeaza functionalitatile de baza necesare control-relor de tip butoane, incat orice programator poate sa isi defineasca propriile clase derivate din aceasta clasa si sa isi creeze butoanele dorite.

In fapt spatiul de nume System.Windows.Forms pune la dispozitie trei control-ere care deriva din clasa ButtonBase, si anume: Button, CheckBox and RadioButton.

Clasa Button. Acest control-er este butonul standard dreptunghiular, care exista pe orice fereastra de dialog. O instanta a clasei Button este utilizata pentru a realiza una din urmatoarele sarcini:

-inchide o fereastra de dialog (spre exemplu, butoanele OK sau Cancel)

-actioneaza asupra unor date introduse intr-o fereastra (spre exemplu butonul Search dupa ce au fost introduse anumite criterii de cautare)

-deschide o fereastra sau o aplicatie (spre exemplu, butonul Help)

Utilizarea unui button este o operatie foarte simpla. In mod uzual consta in adaugarea control-er form-ului nostru si introducerea codului de tratare a evenimentului asociat button-ului, eveniment care in majoritatea cazurilor este Click. Mai jos sunt prezentate cateva dintre proprietatile si evenimentele asociate unui button.

Proprietati (Button):

Nume	Descriere
FlatStyle	The style of the button can be changed with this property. If you set the style to Popup, the button will appear flat until the user moves the mouse pointer over it. When that happens, the button pops up to its normal 3D look.
Enabled	We'll mention this here even though it is derived from Control, because it's a very important property for a button. Setting the Enabled property to false means that the button becomes grayed out and nothing happens when you click it.

Image	Allows you to specify an image (bitmap, icon, etc.), which will be displayed on the button.
ImageAlign	With this property, you can set where the image on the button should appear.

Evenimente (Button):

Cel mai important eveniment este Click. Acesta se produce atunci cand utilizatorul face click pe button. De asemenea, evenimentul este lansat si atunci cand button-ul este focalizat (is focused) si se apasa tasta Enter.

Clasele RadioButton si CheckBox

Desi sunt derivate din aceeasi clasa ca si clasa Button, instantele acestor clase difera substantial atat prin modul in care apar intr-un form cat si prin modul lor de utilizare fata de un button.

Un radioButton, in mod traditional, apare ca un label cu un mic cerculet in stanga, care poate fi selectat sau nu. Aceste butoane (radioButtons) sunt utilizate atunci cand se doreste oferirea posibilitatii utilizatorului de a alege o optiune dintr-o multime de optiuni. Pentru a grupa mai multe radioButton-uri, se utilizeaza in prealabil un GroupBox. Se plaseaza mai intai un GroupBox in form-ul considerat, iar apoi se adauga in interiorul GroupBox-ului atatea radioButton-uri cate sunt necesare.

Un checkBox, in mod traditional, apare ca un label cu un mic dreptunghi in fata care poate avea un semn in interiorul sau. Un checkBox se utilizeaza atunci cand se doreste ca utilizatorul sa aleaga o optiune.

Proprietati (RadioButton):

Nume	Descriere
Appearance	A RadioButton can be displayed either as a label with a circular check to the left, middle or right of it, or as a standard button. When it is displayed as a button, the control will appear pressed when selected and not pressed otherwise.

AutoCheck	When this property is true, a check mark is displayed when the user clicks the radio button. When it is false the radio button must be manually checked in code from the Click event handler.
CheckAlign	By using this property, you can change the alignment of the check box portion of the radio button. The default is ContentAlignment.MiddleLeft.
Checked	Indicates the status of the control. It is true if the control has a check mark, and false otherwise.

Evenimente (RadioButton):

Nume	Descriere
CheckedChanged	This event is sent when the check of the RadioButton changes.
Click	This event is sent every time the RadioButton is clicked. This is not the same as the CheckedChange event, because clicking a RadioButton two or more times in succession only changes the Checked property once -and only if it wasn't checked already. Moreover, if the AutoCheck property of the button being clicked is false, the button will not get checked at all, and again only the Click event will be sent.

Proprietati (CheckBox):

Nume	Descriere
CheckState	Unlike the RadioButton, a CheckBox can have three states: Checked, Indeterminate, and Unchecked. When the state of the check box is Indeterminate, the control check next to the label is usually grayed, indicating that the current value of the check is not valid or has no meaning under the current circumstances.
ThreeState	When this property is false, the user will not be able to change theCheckState state to Indeterminate. You can, however, still change the CheckState property to Indeterminate from code.

Evenimente (CheckBox):

Nume	Descriere
CheckedChanged	Occurs whenever the Checked property of the check box changes. Note that in a CheckBox where the ThreeState property is true, it is possible to click the check box without changing the Checked property. This happens when the check box changes from checked to indeterminate state.
CheckedStateChanged	Occurs whenever the CheckedState property changes. As Checked and Unchecked are both possible values of the CheckedState property, this event will be sent whenever the Checked property changes. In addition to that, it will also be sent when the state changes from Checked to Indeterminate.

Clasele Label si LinkLabel

Instantele clasei Label sunt cele mai utilizate control-ere dintre toate. Aproape fiecare fereastra contine cel putin un Label. Label-ul este un control-er simplu care isi propune un singur lucru si anume sa afiseze un text pe un form.

Arhitectura .NET include doua control-ere label:

-Label, label-ul standard din Windows;

-LinkLabel, un label similar celui anterior si derivat din acesta, insa se prezinta ca un link Internet.

In mod uzual, nu este necesar nici un eveniment pentru Label-ul standard. In schimb, in cazul LinkLabel-ului este necesar cod suplimentar pentru a permite utilizatorului sa acceseze pagina de internet afisata de Text-ul linkLabel-ului.

Un numar mare de proprietati pot fi setate pentru un Label. Multe dintre acestea sunt derivate din clasa Control, insa unele sunt noi. In urmatoarea lista sunt amintite cateva dintre acestea. Daca nu este precizat altfel, proprietatile sunt comune ambelor control-ere.

Proprietati (Label si LinkLabel):

Nume	Descriere
BorderStyle	Allows you to specify the style of the border around the Label. The default is no border.
DisabledLinkColor	(LinkLabel only) The color of the LinkLabel after the user has clicked it.
FlatStyle	Controls how the control is displayed. Setting this property to Popup will make the control appear flat until the user moves the mouse pointer over the control. At that time, the control will appear raised.
Image	This property allows you to specify a single image (bitmap, icon, etc.) to be displayed in the label.
ImageAlign	(Read/Write) Where in the Label the image is shown.
LinkColor	(LinkLabel only) The color of the link.
TextAlign	Where in the control the text is shown.

Clasa TextBoxBase si clasele sale derivate

Clasa TextBoxBase ofera functionalitatea necesara utilizarii si prelucrarii unui text, precum selectarea unui text, cut si paste dintr-un si intr-un Clipbord si multe alte evenimente.

Din clasa TextBoxBase deriva doua clase, si anume TextBox si RichTextBox. Ambele control-ere preiau textul introdus de utilizator.

Un TextBox se utilizeza de regula atunci cand se doreste introducerea unui text care este necunoscut la momentul design-ului. In egala masura se pot introduce orice caractere, sau se poate forta introducerea doar a unor caractere, spre exemplu doar valori numerice.

Un RichTextBox are multe din proprietatile unui TextBox, insa prezinta si unele aspecte mai divese. Daca scopul principal al unui TextBox este acela de a obtine, de la utilizator, un text scurt, un RichTextBox se utilizeaza pentru a introduce si afisa un text sub un anumit format (spre exemplu bold, underline sau italic). Utilizeaza un format standard pentru text numit Rich Text Format sau RTF.

Amintim cateva dintre proprietatile si evenimentele claselor TextBox si RichTextBox.

Proprietati (TextBox):

Nume	Descriere
CausesValidation	When a control that has this property set to true is about to receive focus, two events are fired: Validating and Validated. You can handle these events in order to validate data in the control that is losing focus. This may cause the control never to receive focus. The related events are discussed below.
CharacterCasing	A value indicating if the TextBox changes the case of the text entered. The possible values are: l Lower: All text entered is converted lower case. 2 Normal: No changes are made to the text. 3 Upper: All text entered is converted to upper case.
MaxLength	A value that specifies the maximum length in characters of any text, entered into the TextBox.
Multiline	Indicates if this is a multiline control, which means it is able to show multiple lines of text. When Multiline property is set to true, you'll usually want to set WordWrap to true as well.
PasswordChar	Specifies if a password character should replace the actual characters entered into a single line TextBox. If the Multiline property is true then this has no effect.
ReadOnly	A Boolean indicating if the text is read only.

ScrollBars	Specifies if a multiline TextBox should display scrollbars.
SelectedText	The text that is selected in the TextBox.
SelectionLength	The number of characters selected in the text. If this value is set to be larger than the total number of characters in the text, it is reset by the control to be the total number of characters minus the value of SelectionStart.
SelectionStart	The start of the selected text in a TextBox.
WordWrap	Specifies if a multiline TextBox should automatically wrap words if a line exceeds the width of the control.

Evenimente (TextBox):

Nume	Descriere
Enter, GotFoccus Leave, Validating Validated LostFocus	These six events occur in the order they are listed here. They are known as "Focus Events" and are fired whenever a control's focus changes, with two exceptions. Validating and Validated are only fired if the control that receives focus has the CausesValidation property set to true. The reason why it's the receiving control that fires the event is that there are times where you do not want to validate the control, even if focus changes. An example of this is if the user clicks a Help button.
KeyDown, KeyPress, KeyUp	These three events are known as "Key Events". They allow you to monitor and change what is entered into your controls. KeyDown and KeyUp receive the key code corresponding to the key that was pressed. This allows you to determine if special keys such as *Shift* or *Control* and *F1* were pressed. KeyPress, on the otherhand, receives the character corresponding to a keyboard key. This means that the value for the letter "a" is not the same as the letter "A". It is useful if you want to exclude a range of characters, for example, only allowing numeric values to be entered.
TextChanged	Occurs whenever the text in the TextBox is changed, no matter what the change.

Proprietati (RichTextBox):

Nume	Descriere
Rtf	This corresponds to the Text property, except that this holds the text in RTF.
SelectedRtf	Use this property to get or set the selected text in the control, in RTF. If you copy this text to another application, for example, Word, it will retain all formatting.
SelectedText	Like SelectedRtf you can use this property to get or set the selected text. However, unlike the RTF version of the property, all formatting is lost.
SelectionAlignment	This represents the alignment of the selected text. It can be Center, Left, or Right.
SelectionBullet	Use this property to find out if the selection is formatted with a bullet in front of it, or use it to insert or remove bullets.
BulletIndent	Use this property to specify the number of pixels a bullet should be indented.
SelectionColor	Allows you to change the color of the text in the selection.
SelectionFont	Allows you to change the font of the text in the selection.
SelectionLength	Using this property, you either set or retrieve the length of a selection.
ShowSelectionMargin	If you set this property to true, a margin will be shown at the left of the RichTextBox. This will make it easier for the user to select text.
SelectionProtected	You can specify that certain parts of the text should not be changed by setting this property to true.

Evenimente (RichTextBox):

Nume	Descriere
LinkedClick	This event is sent when a user clicks on a link within the text.
Protected	This event is sent when a user attempts to modify text that has been marked as protected.
SelectionChanged	This event is sent when the selection changes. If for some reason you don't want the user to change the selection, you can prevent the change

Windows Forms

-Realizarea meniurilor

-Controlerul ToolStrip

-Aplicatii MDI si SDI

Realizarea Meniurilor

Majoritatea aplicatiilor Windows contin meniuri. Ahitectura .NET furnizeaza controlere pentru crearea rapida si simpla a meniurilor.

Controlerele MenuStrip si ContextMenuStrip

Aceste controlere, noi odata cu versiunea .NET 2.0, inlocuiesc controlerele MainMenu si ContextMenu, desi si acestea sunt pastrate pentru motive de compatibilitate. Adaugarea acestora unei ferestre se face la fel ca orice alt controler, cu deosebirea ca se plaseaza atat pe fereastra cat si in josul acesteia.

Daca controlerele MainMenu si ContextMenu se compun din obiecte de tip MenuItem, in cazul controlerelor MenuStrip si ContextMenuStrip itemii care le apartin sunt obiecte din unele clase derivate ale clasei ToolStripItem, si anume ToolStripMenuItem, ToolStripComboBox, ToolStipTextBox si ToolStripSeparator. Un obiect instanta al uneia dintre clasele enumerate mai sus poate la randul sau sa fie parinte pentru un submeniu.

Spre deosebire de meniul MenuStrip, care se plaseaza in partea de sus a ferestrei, ContextMenuStrip reprezinta un meniu de context care isi face aparitia in locul in care se face un click dreapta pe un controler (daca meniul de context a fost legat de controlerul respectiv). Este folosit de obicei pentru un acces rapid spre unii itemi ai meniului principal.

In aplicatia aferenta acestui curs vom exemplifica cum se utilizeaza proprietatilor si evenimentele acestor controlere.

Controlerul ToolStrip

Desi meniurile ofera o mare functionalitate aplicatiilor, unii itemi beneficiaza de o atentie sporita fiind plasati atat in meniu cat si in toolbar. In felul acesta se asigura acces mai rapid la unele operatii utilizate frecvent, precum Open, Save, Undo, Paste, etc.

La fel ca in cazul meniurilor, controlerul ToolStrip este nou in .NET 2.0 si inlocuieste controlerul ToolBar.

Spre exemplu, in cazul aplicatilor Word, o selectie de elemente care compun controlerul poate fi:



Un buton care apartine unui controler ToolStrip (sau ToolBar) contine de regula o imagine fara text, desi este posibil sa avem butoane care sa contina ambele elemente. In imaginea de mai sus avem butoane de primul tip. Butoane care contin text pot fi gasite spre exemplu in Internet Explorer. Daca mouse-ul este indreptat spre un buton atunci acesta afiseaza un mesaj (tooltip) care furnizeaza informatii privind scopul butonului.

Elementele unui controler ToolStrip sunt obiecte instante ale urmatoarelor clase: Button, Label, ComboBox, TextBox, Separator, ProgressBar, SplitButton, DropDownButton).

Se pot seta cateva proprietati ale controlerului, precum pozitia pe ecran, insa fiecare element (Button, Label, etc.) este independent fata de celelalte.

Controlerele SplitButton si DropDownButton contin aceleasi obiecte ca si controlerul MenuStrip (si anume ToolStripMenuItem, ToolStripComboBox, ToolStipTextBox si ToolStripSeparator). Diferenta dintre cele doua controlere, SplitButton si DropDownButton, este aceea ca atunci cand se face click pe controlerul DropDownButton sunt afisati itemii acestuia, spre deosebire de SplitButton care afiseaza itemii atunci cand se actioneaza partea din dreapta a butonului (sageata).

Aplicatii MDI si SDI

In mod traditional, sunt trei tipuri de aplicatii care pot fi programate pentru Windows. Aceste sunt:

-- aplicatii de tip dialog. Acestea se prezinta utilizatorului ca un dialog simplu care asigura functionalitatea aplicatiei. Acestea sunt de regula aplicatii mici, simple, destinate unei singure sarcini care necesita o cantitate minima de date. Un exemplu, il reprezinta Calculatorul care este incorporat in sistemul de operare.

-- aplicatii SDI (Simple Document Interfaces). Acestea se prezinta utilizatorului cu un meniu, unul sau mai multe toolbar-uri si o fereastra in care utilizatorul poate realiza unele sarcini. Aplicatiile de acest tip rezolva o sarcina specifica. Utilizatorul poate incarca un singur document in aplicatia in care lucreaza. De regula, sarcina ce urmeaza a fi realizata este complexa si implica o interactiune continua intre utilizator si aplicatie. Exemple de aplicatii SDI sunt WordPad sau Paint. Doar un singur document poate fi deschis la un moment dat.

-- aplicatii MDI (Multple Document Interfaces). Acestea se prezinta utilizatorului in aceeasi maniera ca si o aplicatie SDI, insa are abilitatea de a pastra mai multe ferestre deschise simultan. Un exemplu il reprezinta Adobe Acrobat Reader.

Construirea aplicatiilor MDI

O aplicatie MDI consta din cel putin doua ferestre. Prima fereastra este un container MDI sau fereasta principala (**MDI container**). O fereastra care poate fi afisata in interiorul containerului este numita “copil MDI” (**MDI child**).

Pentru a crea o aplicatie MDI se incepe la fel ca in cazul oricarei aplicatii windows. Pentru a schimba fereastra principala dintr-un form intr-un container MDI trebuie setata proprietatea IsMDIcontainer la true. Se observa cum background-ul ferestrei isi modifica culoarea pentru a marca faptul ca nu ar trebui sa fie plasate controlere pe aceasta, desi este posibila aceasta operatie si chiar rezonabila in unele circumstante.

Pentru a crea o fereastra **MDI child** trebuie adaugata o noua fereastra proiectului prin alegerea unui Windows Form in dialogul obtinut prin selectarea Project | Add New Item. Aceasta fereasta devine un **MDI child** daca este setata proprietatea sa MdiParent incat sa refere fereasta **MDI container**. Aceasta proprietate nu poate fi setata insa din panel-ul Properties ci trebuie setata scriind cod. Doua chestiuni mai raman de facut inainte ca aplicatia sa poata rula in modul convenit. Si anume trebuie ca fereastra **MDI container** sa stie care fereastra **MDI child** sa afiseze iar apoi sa o afiseze. Aceste chestiuni se realizeaza creand o instanta a form-ului care se doreste a fi afisat, iar pentru aceasta se apeleaza metoda Show().

Exemplu:

Daca spatiul de nume (numele proiectului se numeste WindowsFormsApplication6, form-ul **MDI container** se numeste frmContainer, iar form-ul **MDI child** are numele frmChild atunci se parcurg urmatorii pasi:

1. Se modifica constructorul clasei frmChild astfel:

```
public frmChild(WindowsFormsApplication6.frmContainer parent)
 {
    InitializeComponent();
    this.MdiParent = parent;
 }
```

2. Se modifica constructorul clasei frmContainer astfel:

```
public frmContainer()
 {
    InitializeComponent();
    WindowsFormsApplication6.frmChild child = new frmChild(this);
    child.Show();
 }
```

Generice

- Introducere (scop si beneficii)
- Parametrii generici
- Constrangeri asupra parametrilor generici
- Clase generice
- Interfete generice
- Metode Generice
- Delegari generice
- Clasa Nullable

Introducere (Scop si beneficii)

- Genericele (Generics) au fost adaugate odata cu versiunea 2.0 a limbajului C# si motorului comun de programare (CLR).
- Prin conceptul de parametru generic (numit si parametru de tip) este posibila realizarea unei clase (sau metode) care nu specifica unul sau mai multe tipuri utilizate pana la momentul cand clasa (sau metoda) este declarata si instantiata.
- Spre exemplu, prin utilizarea unui parametru generic de tip T, se poate crea o clasa pe care o alta portiune de cod sa o utilizeze fara a suporta costurile operatiilor de impachetare sau despachetare:

```
public class ListaGenerica<T> // Declaram clasa generica.
{    void Add(T input)  { //codul metodei Add
      }
 }
   class Test {
      private class A { }
      static void Main(){
         ListaGenerica<int> lista1 = new ListaGenerica<int>(); // Declaram o lista de tip int.
         ListaGenerica<string> lista2 = new ListaGenerica<string>(); // Declaram o lista de tip string.
         ListaGenerica<A> lista3 = new ListaGenerica<A>(); // Declaram o lista de tip A.
      }
   }
```

-Clasele si metodele generice combină conceptele: reutilizare, tipuri sigure (type safety) și eficiență într-un mod pe care clasele non-generice nu-l pot realiza.

-Generice sunt cel mai frecvent utilizate cu colecțiile și metodele care operează asupra lor. Versiunea . NET 2.0 oferă un nou spațiu de nume, System.Collections.Generic, care conține mai multe clase colectii care utilizeaza genericele.

-Pentru crearea unor tipuri sigure, se recomanda utilizarea colectiilor generice in locul colectiilor non-generice.

-Următorul program prezinta un exemplu simplu in care se poate crea o lista generica. Insa, in cele mai multe cazuri, ar trebui utilizata clasa Lista <T> furnizata de Biblioteca .NET.

```
using System.Collections.Generic;
// pentru parametrul generic T se folosesc
    paranteze unghiulare.
public class ListaGenerica<T>{
    // O clasa imbricata este si ea
    generica in T.
    private class Imbricat{

        private Imbricat next;
        // T ca membru privat.
        private T data;

        public Imbricat(T t){
            next = null;
            data = t;
        }
        public Imbricat Next {
            get { return next; }
            set { next = value; }
        }
        // T ca tip rezultat al unei
    proprietati.
        public T Data{
            get { return data; }
            set { data = value; }
        }
    }
private Imbricat head;
    // constructor
    public ListaGenerica(){
        head = null;
    }
    // T ca tip al unui parametru formal
    pentru o metoda.
    public void AddHead(T t){
        Imbricat n = new Imbricat(t);
        n.Next = head;
        head = n;
    }
    public IEnumerator<T> GetEnumerator() {
        Imbricat current = head;
        while (current != null)
        {      /*Cuvantul cheie yield se
    foloseste intr-o declaratie
              pentru a indica faptul  că
    metoda, operatorul,
               sau accesorul get
               în care apare este un
    iterator.*/
              yield return current.Data;
            current = current.Next;
        }
    }
}
```

```
public class A {
    int i;
    public A(int i){
    this.i = i;
    }
    public override string ToString(){
    return i.ToString();
    }
}
class TestListaGenerica{
    static void Main(){
        // A este tipul argumentului
        ListaGenerica<A> lista =new ListaGenerica<A>();

        for (int x = 0; x < 10; x++){
            lista.AddHead(new A(x));
        }

        foreach (A a in lista)        {
            System.Console.Write(a + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}
```

-Genericele sunt importante la crearea si utilizarea colectiilor generice. Ele permit crearea unei colectii “type-safe” la momentul compilarii.
-Limitele colectiilor non-generice pot fi demonstrate prin urmatorul exemplu, care utilizeaza o colectie ArrayList pentru a pastra obiecte de orice tip:

```
 using System.Collections;
class Demo{
   public static void Main() {
      ArrayList lista1 = new ArrayList();
      lista1.Add(3);
      lista1.Add(105);
      ArrayList lista2 = new System.Collections.ArrayList();
      lista2.Add("Primul string in colectie");
      lista2.Add("Al doilea string in colectie");
   } }
```

- Dar acest confort are un cost. Orice tip referință sau tip valoric, care se adaugă la un ArrayList este implicit transformat intr-un obiect de tipul Object. În cazul în care elementele sunt tipuri valorice, acestea sunt impachetate atunci când sunt adăugate in lista, și despachetate când sunt preluate. Operatiile de impachetare si despachetare scad performanta, iar efectul poate fi foarte important în scenarii care utilizeaza colecții mari.

-Altă limitare conduce la lipsa unei verificarei a codului la momentul compilarii. Nu exista nici un mod de a preintampina codul clientului sa faca ceva de genul acesta:

```
ArrayList listaNoua = new ArrayList();
listaNoua.Add(3);  // Adauga un intreg in lista.
listaNoua.Add("Primul string in colectie."); // Adauga un string in lista.

int t = 0;
// Codul de mai jos lanseaza o exceptie de tipul InvalidCastException.
foreach (int x in listaNoua)
{
   t += x;
}
```

-Deși este perfect acceptabil, și poate ca uneori în mod intenționat doriți să creați o colecție eterogenă, combinarea stringurilor si intregilor intr-un singur ArrayList este mult mai probabil să fie o eroare de programare, iar această eroare nu va fi detectata până la execuție (se lanseaza o exceptie).

-ArrayList și alte clase similare au nevoie de o modalitate prin care codul utilizatorului sa specifice tipul de date special pe care aceste clase intentioneaza sa-l foloseasca. Aceasta ar elimiana nevoia de utilizare a castului si ar facilita calea compilatorului să facă verificare a tipului.
-Cu alte cuvinte, ArrayList are nevoie de un parametru care sa specifice tipul elementelor din colectie. Asta este exact ceea ce oferă genericele. În colecția generica Lista <T>, din spatiul de nume System.Collections.Generic, aceeași operațiune de adăugarea de elemente unei colectii seamănă cu aceasta:

```
List<int> lista1 = new List<int>();
  // Fara impachetare sau cast:
  lista1.Add(3);
  // Eroare la compilare:
  // lista1.Add("Un string de adaugat in colectie.");
```

- Pentru codul client, sintaxa de adăugat atunci cand se utilizeaza List <T> in detrimentul unui ArrayList este argumentul <T> atat in declaratie cat si in instantiere. În schimb pentru această mica extindere, puteți crea o listă, care nu este doar mai sigura decât ArrayList, dar, de asemenea, semnificativ mai rapida, mai ales atunci când elementele din listă sunt tipuri valorice.

Parametrii generici

-Un parametru generic (sau parametru de tip) reprezinta un substituent pentru un anumit tip care urmeaza a fi specificat de utilizator atunci când instantiaza o variabilă de tip generic.
-O clasă generica, cum ar fi ListaGenerica <T> utilizata intr-un slide anterior, nu poate fi utilizata ca atare, pentru că nu este într-adevăr un tip, este mai mult un plan pentru un tip.
-Pentru a utiliza ListaGenerica <T>, codul client trebuie să declare şi instantieze un tip prin specificarea unui argument în interiorul parantezelor unghiulare. Argumentul pentru această clasă particulară poate fi orice tip recunoscut de compilator. Exemplu:

```
ListaGenerica<int> lista1 = new ListaGenerica<int>();
ListaGenerica<string> lista2 = new ListaGenerica<string>();
ListaGenerica<A> lista3 = new ListaGenerica<A>();
```

-În fiecare din aceste instante ale clasei GenericList <T>, fiecare apariție a parametrului T în clasa va fi înlocuit în timpul rulării cu argumentul specificat. Prin această înlocuire, am creat trei obiecte de tipuri diferite utilizând o singură definiție de clasă.

Numele parametrilor generici:

-Denumiti parametrii generici cu nume descriptiv, cu exceptia cazului cand o singura litera este auto-explicativa si un nume ar adăuga nimic in plus. Exemple:

```
public interface IDictionary<TKey,TValue>
public delegate TOutput Converter<TInput, TOutput>( TInput input)
```

-Considerati utilizarea literei T ca numele parametrului pentru tipurile cu un singur parametru. Exemple:

```
public int IComparer<T>() { return 0; }
public delegate bool Predicate<T>(T item);
public struct Nullable<T> where T : struct { /*...*/ }
```

Constrangeri asupra parametrilor generici

- Când definiți o clasă generica, se pot aplica restricții asupra tipurilor pe care codul client le poate utiliza drept argumente de tip atunci cand se instanțiază clasa generica.
- În cazul în care codul client încearcă să instantieze clasa generica, prin utilizarea unui tip care nu este permis, rezultatul este o eroare de compilare. Aceste restricții sunt numite constrângeri.
- Constrângerile sunt specificate prin folosirea cuvantului cheie where.

Următorul tabel listează cele șase tipuri de constrângeri:

Constrangere	Descriere
where T: struct	Argumentul de tip trebuie sa fie un tip valoric. Asadar T poate fi orice tip valoric cu exceptia tipului Nullable.
where T : class	Argumentul T trebuie sa fie un tip referinta; aceasta se aplica pentru orice clasa, interfata, delegare sau tablou.
where T : new()	Argumentul de tip trebuie sa aiba un constructor public fara nici un parametru. Atunci cand se utilizeaza cu mai multe constrangeri, new() trebuie specificat ultimul.
where T : <base class name>	Argumentul T trebuie sa fie de tipul clasei sau sa derive din clasa specificata.
where T : <interface name>	Argumentul T trebuie sa fie sau sa implementeze interfata. Se pot include constrangeri interfata multiple. Contrangerea interfata poate fi de asemenea generica.
where T : U	Argumentul T trebuie sa fie de tipul sau sa derive din argumentul specificat pentru U.

De ce se utilizeaza constrangerile?

- Dacă doriți a examina un element dintr-o listă generică pentru a determina dacă acesta este valid sau nu sau pentru a-l compara cu un alt element, compilatorul trebuie să aibă o oarecare garanție că metoda pe care o utilizeaza este suportata de către orice tip de argument care ar putea fi specificat de codul client. Această garanție este obținută prin aplicarea uneia sau mai multor constrângeri în definiția clasei generice.

-De exemplu, constrângerea "clasa de baza" spune compilatorului că numai obiecte de acest tip sau derivate din acest tip vor fi utilizate ca argumente de tip. Odata ce compilatorul are această garanție, se poate permite metodei de acest tip să apeleze clasa generica.

- Clasa ListaGenerica <T> o putem modifica astfel:

```
public class ListaGenerica<T> where T:A
{
}
```

Clase generice

-Clasele generice includ operațiuni care nu sunt specifice pentru un tip particular de date.

- Cea mai comună utilizare a claselor generice este in colecții, cum ar fi: liste, stive, cozi etc. Operații cum ar fi adăugarea sau eliminarea de elemente dintr-o colectie sunt efectuate în esență in același mod, indiferent de tipul de date stocate.

- Pentru cele mai multe scenarii care necesită clase colectii, se recomandă utilizarea claselor prevăzute în Biblioteca .NET.

-De obicei, creați clase generice plecand de la o clasa concreta (non-generica) si schimband pe rand tipurile dorite in tipuri generice pana se atinge un echilibru optim intre generalizare si usurinta in utilizare.

Când creati propriile clase generice, considerati importante următoarele considerente:

-*Ce tipuri ar trebui generalizate in parametrii generici?*

Ca o regulă, cu cat mai multe tipuri puteți parametrizarea cu atat codul devine mai flexibil și reutilizabil. Cu toate acestea, o generalizare prea extinsa poate crea cod care este dificil pentru alți dezvoltatori sa-l citeasca sau sa-l înțeleaga.

-*Ce constrângeri, dacă este cazul, să se aplice parametrilor generici?*

O regulă bună este să se aplice constrângeri maxime posibile, care vor permite în continuare manipularea tipurilor pe care aplicatia (programul) le are in vedere. De exemplu , dacă știți că acea clasa generica este destinata utilizarii numai cu tipurile referință, se aplică restricția de clasă. Aceasta va preintampina utilizarea clasei pentru tipuri valorice si facilita folosirea operatorului as si testarea valorilor null.

-*Dacă se impune utilizarea comportamentului generic in clase si subclase.*

Deoarece clasele generice poate servi drept clase de bază, aceleași considerente de proiectare se aplică aici ca si in cazul claselor non- generice.

-*Daca se impune implementarea unei (sau mai multor) interfețe generice.*

De exemplu, dacă proiectați o clasă care va fi folosita pentru a crea elemente ale unei colectii generice, va trebui să implementati o interfață, cum ar fi IComparable <T> unde T este tipul clasei dumneavoastra.

Regulile pentru parametrii genericii și pentru constrângeri au mai multe implicații asupra comportamentului clasei generice, în special ceea ce privește moștenirea și accesibilitate membrilor. Astfel:

- Pentru o clasa generica ClasaMeaGenerica <T>, codul client poate face referire la clasa, fie prin specificarea unui argument de tip, pentru a crea un **tip închis** (ClasaMeaGenerica <int>). Alternativ, se poate lăsa parametrul de tip nespecificat, de exemplu, atunci când specificați o clasa de baza generica, pentru a crea un **tip deschis** (ClasaMeaGenerica <T>) . Clase generice pot moșteni o clasa concreta, un tip inchis sau un tip deschis:

```
class ClasaMea { }
class ClasaMeaGenerica<T> { }

// clasa generica mosteneste clasa nongenerica (concreta)
class MostenesteClasaConcreta<T> : ClasaMea { }
// clasa generica mosteneste clasa generica inchisa
class MostenesteClasaInchisa<T> : ClasaMeaGenerica<int> { }
//clasa generica mosteneste clasa generica deschisa
class MostenesteClasaDeschisa<T> : ClasaMeaGenerica<T> { }
```

-Clasele non-generice (concrete) pot moșteni tipuri închise, dar nu tipuri deschise sau parametri de tip deoarece în timpul rulării nu există nici o modalitate pentru codul client de a furniza argumentul de tip necesar pentru a instantia clasa de baza:

```
class ClasaMea : ClasaMeaGenerica<int> { } //fara eroare
//class ClasaMea : ClasaMeaGenerica<T> { } //Genereaza o eroare
//class ClasaMea : T {} //Genereaza o eroare
```

- Clasele generice care moștenesc tipuri deschise trebuie să furnizeze argumente de tip pentru fiecare parametru de tip al clasei de baza care nu intervine explicit în clasa derivata, așa cum se arata în următorul cod:

```
class ClasaMeaGenerica<T, U> { }
class ClasaA<T> : ClasaMeaGenerica<T, int> { } //fara eroare
class ClasaB<T, U> : ClasaMeaGenerica<T, U> { } //fara eroare
//class ClasaC<T> : ClasaMeaGenerica<T, U> {}  //genereaza eroare
```

-Clasele generice care moștenesc tipuri deschise trebuie să implice constrângerile tipului:

```
class ClasaMeaGenerica<T> where T : System.IComparable<T>, new() { }

class ClasaMeaGenericaSpeciala<T> : ClasaMeaGenerica<T> where T :
System.IComparable<T>, new() { }
```

-Tipurile generice pot utiliza parametrii de tip multiplii și impune constrângeri multiple, după cum urmează:

```
class ClasaMeaGenerica<K, V, U>
    where U : System.IComparable<U>
    where V : new()
  { }
```

-Tipurile deschise sau inchise pot fi utilizate ca parametrii pentru metode:

```
void MetodaMea<T>(List<T> lista1, List<T> lista2)
  {     //codul metodei    }

void MetodaMea(List<int> lista1, List<int> lista2)
  {     //codul metodei    }
```

Interfete generice

Adesea este util să se definească interfețe, fie pentru clase colectii generice, sau pentru clase generice care reprezintă elemente intr-o colecție.

De preferat, atunci cand se utilizeaza clase generice, este de a utiliza interfețe generice, cum ar fi IComparable <T> in locul lui IComparable, aceasta pentru a evita operatiile de impachetare si despachetare a tipurilor valorice.

Biblioteca .NET defineste mai multe interfețe generice pentru a fi utilizate de clasele colectii din spațiul de nume System.Collections.Generic.

Atunci când o interfață este specificata ca o constrângere asupra unui parametru de tip, pot fi utilizate numai tipuri care implementeaza interfata (a se vedea exemplul urmator unde intr-o lista de tipul ListaSortata<T> pot fi adaugate doar instante ale unor clase care implementeaza interfata System.IComparable<T>.

```
using System.Collections.Generic;
using System.Collections:

public class ListaGenerica<T>: IEnumerable<T>{
    protected Imbricat head;
    // O clasa imbricata este si ea generica in T.
   protected class Imbricat{
       private Imbricat next;
      // T ca membru privat.
      private T data;

      public Imbricat(T t){
         next = null;
         data = t;
      }
      public Imbricat Next {
         get { return next; }
         set { next = value; }
      }
      // T ca tip rezultat al unei proprietati.
      public T Data{
         get { return data; }
         set { data = value; }
      }
   }
    public ListaGenerica(){
       head = null;
    }
    public void AddHead(T t){
       Imbricat n = new Imbricat(t);
       n.Next = head;
       head = n;
    }
   public IEnumerator<T> GetEnumerator()  {
       Imbricat current = head;
       while (current != null)   {
           yield return current.Data;
           current = current.Next;
       }   }
   // IEnumerable<T> mosteneste interfata IEnumerable din
   //System.Collections,
   // asadar aceasta clasa trebuie sa implementeze ambele
   // versiuni generica si negenerica
   // ale metodei  GetEnumerator. In cele mai multe cazuri
   // metoda negenerica
   // face apel la metoda generica.
    IEnumerator IEnumerable.GetEnumerator()   {
       return GetEnumerator();
    }
}
```

```
public class ListaSortata<T> : ListaGenerica<T> where
      T : System.IComparable<T>
{
   //Un algoritm de sortare a elementelor
   //de la cel mai mic la cel mai mare

   public void MetodaSortare()
   {
      if (null == head || null == head.Next)
      {
         return;
      }
      bool schimb;

      do
      {
         Imbricat previous = null;
         Imbricat current = head;
         schimb = false;

         while (current.Next != null)
         {
            if (current.Data.CompareTo(current.Next.Data) > 0) {
                  Imbricat tmp = current.Next;
                  current.Next = current.Next.Next;
                  tmp.Next = current;

                  if (previous == null)  {
                     head = tmp;
                  }
                  else  {
                     previous.Next = tmp;
                  }
                  previous = tmp;
                  schimb = true;
            }
            else {
                  previous = current;
                  current = current.Next;
            }
         }
      } while (schimb);
   }
}
```

```
public class A: System.IComparable<A>
{
   int i;
   public A(int i)
   {
      this.i = i;
   }

   public int CompareTo(A a)
   {
      return i - a.i;
   }

   public override string ToString()
   {
      return i.ToString();
   }
}
```

```
class TestListaGenerica{
   static void Main()  {
      // A este tipul argumentului
      ListaSortata<A> lista = new ListaSortata<A>();

      for (int x = 0; x < 10; x++)  {
         lista.AddHead(new A(x));
      }

      foreach (A a in lista)    {
         System.Console.Write(a + " ");
      }
      System.Console.WriteLine();

      lista.MetodaSortare();

      foreach (A a in lista)
      {
         System.Console.Write(a + " ");
      }
      System.Console.WriteLine("\nDone");
   }
}
```

-Interfețe multiple pot fi specificate drept constrângeri asupra unui singur tip, după cum urmează:

```
class Stack<T> where T : System.IComparable<T>, IEnumerable<T> { }
```

-O interfață poate defini mai mult de un parametru de tip, după cum urmează:

```
interface IDictionary<K, V> { }
```

-Regulile de moștenire care se aplică la clase, se aplică de asemenea la interfețe:

```
interface ILuna<T> { }
interface IJanuarie: ILuna<int> { } //fara eroare
interface IFebruarie<T> : ILuna<int> { } // fara eroare
interface IMartie<T> : ILuna<T> { } // fara eroare
//interface IAprilie : ILuna<T> {} //eroare
```

Obs: Interfețele generice pot moșteni interfețe non-generice, doar dacă interfața generica este contra-variantă, ceea ce înseamnă că isi folosește parametrul de tip ca o valoare return. În biblioteca .NET, IEnumerable <T> moștenește IEnumerable deoarece IEnumerable <T> folosește T ca valoare return a metodei GetEnumerator.

-Clasele concrete pot implementa interfețe închise, după cum urmează:

```
interface IInterfataDeBaza<T> { }
class ClasaDemo : IInterfataDeBaza<string> { }
```

-Clasele generice pot implementa interfețe generice sau interfețe închise atâta timp cât lista de parametri tip ai clasei cuprinde toate argumentele necesare interfeței, după cum urmează:

```
interface IInterfataDeBaza1<T> { }
interface IInterfataDeBaza2<T, U> { }

class ClasaDemo1<T> IInterfataDeBaza1<T> { } //fara eroare
class ClasaDemo2<T> : IInterfataDeBaza2<T, string> { } //fara eroare
```

Metode Generice

-O metodă generică este o metodă care este declarată cu parametrii de tip, după cum urmează:

```
static void Schimb<T>(ref T stanga, ref T dreapta)
{ T temp;
temp = stanga;
stanga = dreapta;
dreapta = temp; }
```

-Următorul exemplu arată o modalitate de a apela metoda folosind int pe post de argument de tip:

```
public static void TestSchimb() {
   int a = 1; int b = 2;
   Schimb<int>(ref a, ref b);
   System.Console.WriteLine(a + " " + b);
 }
```

-Puteți omite argumentul de tip intrucat compilatorul il va deduce. Următorul apel pentru Schimb este echivalent cu apelul anterior:

```
Schimb(ref a, ref b);
```

Obs. Aceleași reguli de deducere a tipului la apelul metodei, daca tipul nu este precizat (vezi slide-ul anterior) se aplică atat metodelor statice cat și metodelor instantelor. Compilatorul poate deduce parametrii de tip (T in slide-ul anterior), pe baza argumentelor primite de metoda. Compilatorul nu poate deduce parametrii de tip daca metoda nu are parametrii formali. Prin urmare deducerea tipului funcționează cu metode care au parametri formali de tipul T.

-Într-o clasă generică, metodele non-generice pot accesa parametrii de tip (ai clasei), după cum urmează:

```
class ClasaDemo<T> {
        void OMetoda(ref T lhs, ref T rhs) { }
}
```

-Dacă definiți o metodă generică care are aceiași parametri de tip ca si clasa care conține metoda, compilatorul generează avertismentul CS0693, deoarece în blocul metodei, argumentul furnizat de T interior ascunde argumentul furnizat de T exterior.

Dacă aveți nevoie de flexibilitate in apelul unei metode generice cu argumente de tip, altele decât cele furnizate clasei atunci cand aceasta a fost instantiata, atunci se ia în considerare furnizarea unui alt identificator pentru parametrul tip al metodei, așa cum se arată în ListaGenerica2 <T>:

```
class ListaGenerica1<T>
{
        // generaza avertismentul CS0693
        void MetodaDemo<T>() { }
}
class ListaGenerica2<T>
{
        //nici un avertisment
        void MetodaDemo<U>() { }
}
```

-Utilizați constrângeri, pentru a permite operațiuni specializate asupra parametrilor de tip ai metodelor. Această metoda numita Schimb2 <T>, poate fi utilizata numai cu argumente de tip care implementeaza IComparable <T>.

```
void Schimb2<T>(ref T stanga, ref T dreapta) where T : System.IComparable<T>
        {
                T temp;
                if (stanga.CompareTo(dreapta) > 0)
                {
                        temp = stanga;
                        stanga = dreapta;
                        dreapta= temp;
                }
        }
```

-Metodele generice poate fi supraîncărcate pe mai mulți parametri de tip. De exemplu, următoarele metode pot fi situate în aceeași clasă:

```
void  Metoda() { }
void  Metoda<T>() { }
void Metoda<T, U>() { }
```

Delegari generice

-O delegare poate defini proprii sai parametri tip. Codul care refera delegarea generica poate specifica argumentul de tip pentru a crea un tip închis, la fel ca atunci când se instantiaza o clasă generica sau se apeleaza o metodă generica, așa cum se arată în următorul exemplu:

```
public delegate void Del<T>(T item);
public static void Metoda(int i) { }

Del<int> m1 = new Del<int>(Metoda);
```

-Versiunea C# 2.0 are o nouă caracteristică, care functioneaza atat cu delegarile concrete, precum și cu delegarile generice, și vă permite să scrieti ultima linie de mai sus cu această sintaxă simplificată:

```
Del<int> m2 = Metoda;
```

-Delegarile definite într-o clasă generica pot utiliza parametrii de tip clasa generica în acelaşi mod în care o fac metodele clasei:

```
class Demo<T> {
        T[ ] items;
        int index;
        public delegate void DelegareDemo(T[ ] items);
}
```

-Codul care face referire la delegare trebuie să specifice argumentul tip al clasei continute, după cum urmează:

```
private static void Metoda(float[] items) { }

public static void TestMetoda() {
        Demo<float> s = new Demo<float>();
Demo<float>.DelegareDemo d = Metoda;
}
```

Clasa Nullable

-Tipurile valorice difera de tipurile referinta prin faptul ca primele contin o valoare. Tipurile valorice pot exista in stare sa zicem “neatribuita” imediat dupa ce sunt declarate si inainte de a li se atribui o valoare. Insa nu pot fi utilizate intr-o expresie daca nu li se atribuie o valoare.

-Din contra, un tip referinta poate fi null.

-Exista cazuri cand este necesar sa avem o valoare pentru orice tip folosit, chiar daca aceasta este null (in particular cand se lucreaza cu baze de date).

-Genericele ofera o modalitate de a face acest lucru prin utilizarea clasei generice System.Nullable<T>, spre exemplu:

```
System.Nullable<int>    nullableInt;
```

-Acest cod declara o variablila care poate avea orice valoare de tip int insa si valoarea null.

-Se poate scrie

```
nullableInt=null;
```

cod echivalent cu

```
nullableInt=new System.Nullable<int>();
```

- Clasa Nullable<T> pune la dispozitie proprietatile HasValue si Value. Daca HasValue este true atunci este garantata o valoare pentru Value. In caz contrar (HasValue este false), daca se apeleaza Value atunci este lansata o exceptie de tipul System.InvalidOperationException.

- Intrucat tipurile Nullable<T> sunt des utilizate, in locul sintaxei:

```
System.Nullable<int> nullableInt;
```

se utilizeaza forma prescurtata:

```
int? nullableInt;
```

-In cazul tipurilor simple precum int, se pot utiliza operatorii +, -, * etc. pentru a lucra cu valori. In cea ce priveste tipurile Nullable nu exista nici o diferenta. Spre exemplu, putem avea:

```
int? op1=5;
int? result=op1*2;
```

Rezultatul este de tip int?

-Insa urmatorul cod nu poate fi compilat:

```
int? op1=5;
int  result=op1*2;
```

-Pentru ca lucrurile sa fie in ordine, trebuie utilizat un cast:

```
int? op1=5;
int  result=(int)op1*2;
```

sau utilizata proprietatea Value

```
int? op1=5;
int  result=op1.Value*2;
```

-Ce se intampla cand unul din operanzi este null?

Raspunsul este urmatorul: pentru toate tipurile simple Nullable, in afara de bool?, rezultatul este null. Pentru bool? rezultatul este dat in tabelul

op1	op2	op1 &op2	op1 |op2
true	null	null	true
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

Exemplu (clasa Vector):

```
using System;
public class Vector
{
    public double? r = null;
    public double? theta = null;

    public double? ThetaRadiani
    {
        get
        {
            //theta in radiani
            return theta * 4 * Math.Atan(1) / 180;
        }
    }
    public Vector(double? r, double? theta)
    {
        theta = theta % 360;

        this.theta = theta;
        this.r = r;
    }
```

```
 public static Vector operator +(Vector op1, Vector op2)    {
        try{
            double X_suma = op1.r.Value * Math.Cos(op1.ThetaRadiani.Value) + op2.r.Value *
Math.Cos(op2.ThetaRadiani.Value);
            double Y_suma = op1.r.Value * Math.Sin(op1.ThetaRadiani.Value) + op2.r.Value *
Math.Sin(op2.ThetaRadiani.Value);
            double r_suma = Math.Sqrt(X_suma*X_suma+Y_suma*Y_suma);
            double theta_suma = Math.Atan2(Y_suma, X_suma)*180/4/Math.Atan(1);
            return new Vector(r_suma, theta_suma);
        }
        catch{
            return new Vector(null, null);
        }
    }
    public static Vector operator -(Vector op)   {
        Vector v = new Vector(null, null);
        v.r = op.r;
        v.theta = op.theta + 180;
        return v;
    }

    public static Vector operator -(Vector op1, Vector op2)   {
        return op1 + (-op2);
    }
    public override string ToString()    {
        string r_string = r.HasValue ? r.ToString() : null;
        string theta_string = theta.HasValue ? theta.ToString() : null;
        return string.Format("{0}[cos({1})+i sin({1})]", r_string, theta_string);
    }
}
```

```
class Program{
    public static void Main()   {
        Vector v1 = ObtineVector("vectorul1");
        Vector v2 = ObtineVector("vectorul2");
        Console.WriteLine("{0}+{1}={2}", v1, v2, v1+v2);
        Console.WriteLine("{0}-{1}={2}", v1, v2, v1 - v2);
        Console.ReadKey();
    }
    public static Vector ObtineVector(string numeVector)   {
        Console.Write("Introduceti magnitudinea pentru {0}: ", numeVector);
        double? r = ObtineNullableDouble();
        Console.Write("\n Introduceti unghiul (in grade) pentru {0}: ", numeVector);
        double? theta = ObtineNullableDouble();
        Console.Write("\n");
        return new Vector(r,theta);
    }
    public static double? ObtineNullableDouble()   {
        double? rezultat;
        string string_introdus=Console.ReadLine();
        try  {
            rezultat = double.Parse(string_introdus);
        }
        catch
        {
            rezultat = null;
        }
        return rezultat;
    }
}
```

Colecții generice

Introducere

Clasele spațiului de nume System.Collections.Generic

Structurile spațiului de nume System.Collections.Generic

Interfețele spațiului de nume System.Collections.Generic

Clasa List<T>

Clasa Dictionary<TKey, TValue>

Colecții sortate

Cozi și stive

Introducere

-Datele având aceleași caracteristici pot fi tratate mai eficient atunci când grupate într-o colecție. În loc de a scrie cod care să trateze separat fiecare obiect în parte, prin intermediul colecțiilor puteți utiliza același cod pentru a procesa toate elementele care au caracteristici comune.

-Pentru a gestiona o colecție cu un număr fix de elemente se poate utiliza clasa System.Array. Pentru a adăuga, elimina și modifica fie elemente individuale sau o serie de elemente în colecție se pot utiliza clasele din System.Collections, așa cum am procedat în cursurile anterioare, însă și o mulțime de alte clase din spațiul de nume System.Collections.Generic care permit crearea unor colectii puternic tipizate.

-.NET Framework 4 , adauga spațiul de nume System.Collections.Concurrent care furnizează clase "thread-safe", utile atunci când elementele colecției sunt accesate concurent de mai multe fire de execuție.

 - Clasele din spațiul de nume System.Collections.Immutable permit lucrul cu colectii care nu pot fi schimbate și care prin modificare creaza noi colecții. Aceste clase au fost adaugate de versiunea .NET 4.5 și vizează aplicații care se adresează desktop-ului, Windows Phone 8, Windows Store.

-Colectiile generice au fost adaugate de versiunea .NET 2.0. In acest sens, a fost adăugat spatiul de nume System.Collections.Generic, care conține mai multe clase colectii care utilizeaza genericele.

-Genericele permit crearea unei colectii "type-safe" la momentul compilarii. Daca o colectie ArrayList poate pastra obiecte de orice tip, in schimb intr-o colectie de tipul List<T> se pot pastra doar obiecte de tipul T.

- Orice tip referință sau tip valoric, care se adaugă la un ArrayList este implicit transformat intr-un obiect de tipul Object. În cazul în care elementele sunt tipuri valorice, acestea sunt impachetate atunci când sunt adăugate in lista, și despachetate când sunt preluate. Operatiile de impachetare si despachetare scad performanta, iar efectul poate fi foarte important în scenarii care utilizeaza colecții mari.

-Din motivele mentionate mai sus, pentru crearea unor tipuri sigure, se recomanda utilizarea colectiilor generice in locul colectiilor non-generice.

Clasele spațiului de nume System.Collections.Generic

Clasa	Descriere
Comparer <T>	Clasă care oferă implementări ale interfeței generice IComparer <T> .
Dictionary<TKey, TValue>	Reprezintă o colecție de chei și valori.
Dictionary<TKey, TValue>.KeyCollection	Reprezintă colecția de chei într-un Dictionary<TKey,TValue>. Aceasta clasa nu poate fi moștenită.
Dictionary <TKey, Tvalue>.ValueCollection	Reprezintă colecția de valori într-un Dictionary<TKey,TValue> . Aceasta clasa nu poate fi moștenită .
EqualityComparer <T>	Clasa care oferă implementări ale interfeței generice IEqualityComparer <T> .
HashSet <T>	Reprezintă un set de valori.
KeyedByTypeCollection <TItem>	Oferă o colecție ale cărei elemente sunt tipuri care servesc drept chei.
KeyNotFoundException	Excepția este lansată , atunci când cheia specificată pentru accesarea unui element într-o colecție nu se potrivește cu nici o cheie din colecție.
LinkedList <T>	Reprezintă o listă dublu legată.
LinkedListNode <T>	Reprezintă un nod în LinkedList <T> . Această clasă nu poate fi moștenită.
List <T>	Reprezintă o listă puternic tipizată de obiecte care pot fi accesate de un index. Oferă metode de căutare, sortare și manipulare a listelor
Queue<T>	Reprezintă o colecție first-in, first-out (FILO) de obiecte.

SortedDictionary <TKey, TValue>	Reprezintă o colecție de perechi cheie / valoare care sunt sortate pe baza cheii.
SortedDictionary <TKey, TValue> . KeyCollection	Reprezintă colecția de chei într-un SortedDictionary <TKey, TValue> . Această clasă nu poate fi moștenită.
SortedDictionary <TKey, TValue> . ValueCollection	Reprezintă colecția de valori într-un SortedDictionary <TKey, TValue> . Această clasă nu poate fi moștenită.
SortedList <TKey, TValue>	Reprezintă o colecție de perechi cheie / valoare care sunt sortate prin punerea în aplicare a interfeței IComparer <T> .
SortedSet <T>	Reprezintă o colecție de obiecte care se menține în ordine sortată.
Stack <T>	Reprezintă o colecție last-in first-out (LIFO) de obiecte de același tip.
SynchronizedCollection <T>	Oferă o colecție thread-safe, care conține obiecte de tipul specificat de parametrul generic.
SynchronizedKeyedCollection <K, T>	Oferă o colecție thread-safe, care conține obiecte de tipul specificat de parametrul generic și care sunt grupate pe chei.
SynchronizedReadOnlyCollection <T>	Oferă o colecție thread-safe read-only , care conține obiecte de tipul specificat de parametrul generic.

Structurile spațiului de nume System.Collections.Generic

- Cu exceptia structurii:

KeyValuePair <TKey, TValue> *Definește o pereche cheie/valoare care poate fi setată sau recuperată,*

restul structurilor sunt enumeratori ai claselor spatiului de nume System.Collections.Generic.

- Fiecare din structurile următoare enumeră elementele din clasa corespunzătoare. Spre exemplu, Dictionary <TKey, TValue> .Enumerator enumeră elementele unui Dictionary <TKey, TValue>. (Metoda GetEnumerator din acesta clasa returneaza o structura Dictionary <TKey, TValue> .Enumerator).

Celelalte structuri ale spațiului de nume System.Collections.Generic sunt:

Dictionary <TKey, TValue>. KeyCollection.Enumerator ;
Dictionary <TKey, TValue>. ValueCollection.Enumerator;
HashSet <T>.Enumerator ;
LinkedList <T> .Enumerator ;
List <T> .Enumerator;
Queue <T> .Enumerator;
SortedDictionary <TKey, TValue> . Enumerator;
SortedDictionary <TKey, TValue> . KeyCollection.Enumerator ;
SortedDictionary <TKey, TValue> . ValueCollection.Enumerator;
SortedSet <T> .Enumerator;
Stack <T> . Enumerator;

Interfețele spațiului de nume System.Collections.Generic

Interfața	Descriere
ICollection <T>	Definește metode pentru a manipula colecții generice (Add, Clear, Contains, CopyTo, Remove).
IComparer <T>	Definește o metodă (Compare) pe care un tip trebuie să o implementeze pentru a compara două obiecte de tipul T.
IDictionary <TKey, TValue>	Reprezintă o colecție generică de perechi cheie / valoare.
IEnumerable <T>	Expune enumaratorul, care realizează o iterare simplă a unei colecții de un tip specificat .
IEnumerator <T>	Realizează o iterare simplă a unei colecții generice.
IEqualityComparer <T>	Definește metode pentru compararea obiectelor.
IList <T>	Reprezintă o colecție de obiecte care pot fi accesate individual printr-un index .
IReadOnlyCollection <T>	Reprezintă o colecție read-only puternic tipizată de elemente.
IReadOnlyDictionary <TKey, TValue>	Reprezintă o colecție read-only de perechi cheie/valoare.
IReadOnlyList <T>	Reprezintă o colecție read-only de elemente care pot fi accesate printr-un index .

Clasa List<T>

Reprezintă o listă puternic tipizată de obiecte care pot fi accesate prin intermediul unui index. Clasa furnizează metode pentru căutare, sortare și manipularea unei liste.

Proprietățile clasei List<T>

Capacity	Obține sau setează numărul total de elemente pe care structura internă de date le poate stoca fără redimensionare.
Count	Obține numărul de elemente cuprinse în List <T>.
this[int]	Obține sau setează elementul la indexul specificat (indexarea clasei List<T>).

Câteva metode ale clasei List <T>

Add	Adaugă un obiect la sfârșitul listei List <T>.
AddRange	Adaugă elementele colecției specificate la sfârșitul listei List<T>.
Clear	Elimină toate elementele din List <T>.
Contains	Determină dacă un element este în List <T>.
ConvertAll <TOutput>	Convertește elementele din lista curentă List <T> la un alt tip (tipul TOutput), și returnează o listă care conține elementele convertite.
CopyTo (T [])	Copie întreaga listă List <T> într-un tablou (Array) compatibil unidimensional. Punctul de start este la începutul matricei țintă.
GetEnumerator	Returnează un enumerator (structura List<T>.Enumerator) care iterează lista List <T> . (Metoda MoveNext() din structura List<T>.Enumerator este apelata automat de bucla foreach)
GetRange	Crează o copie superficială a unor elemente din List <T> .
IndexOf (T)	Caută obiectul specificat și întoarce prima apariție (indexarea se face de la zero) în întreaga listă List <T> .

Insert(int, T)	Inserează un element în List <T> la indexul specificat .
InsertRange	Inserează elementele unei colecții în lista List <T> la indexul specificat .
LastIndexOf (T)	Caută obiectul specificat și întoarce indexul ultimei apariții în întreaga listă List<T> .
Remove(T)	Sterge obiectul specificat la prima apariție a sa în List <T> .
RemoveAt(int)	Elimină elementul avand indexul specificat de List <T> .
Reverse ( )	Inversează ordinea elementelor din întreaga listă List <T> .
Sort ( )	Sortează elementele din întreaga listă List <T> folosind comparatorul implicit.
Sort (IComparer <T>)	Sortează elementele din întreaga listă List<T> folosind comparatorul specificat .

Următoarele două programe demonstrează modul în care se poate utiliza clasa List <T> împreună cu proprietățile și metodele sale, însă și alte aspecte generice: clasa Comparer <T>, interfețele IEquatable<T>, IComparable<T> și IComparer<T>.

-Remarcați în primul program cum se pot utiliza metodele Contains(T) și Remove(T) (ICollection<T>) implementate de List <T> în conjuncție cu interfața IEquatable<T>.

-În cel de-al doilea program, remarcați cum se pot utiliza IComparable<T> și IComparer<T> pentru a sorta o colecție. Comparați programul de față cu un program similar prezentat în cursul 9.

Exemplu List<T>

```
using System;
using System.Collections.Generic;

public class Produs : IEquatable<Produs>   {
    public int produs_ID;
    public string numeProdus;

    public Produs(int produs_ID, string numeProdus)  {
        this.produs_ID = produs_ID;
        this.numeProdus = numeProdus;
    }

    public override string ToString()   {
        return "Produs_ID: " + produs_ID + "   Nume: " + numeProdus;
    }

    public bool Equals(Produs altProdus)  {
        if (altProdus == null) return false;
        return (this.produs_ID.Equals(altProdus.produs_ID));
    }
}
```

```
class Exemplu {
    public static void Main() {
        List<Produs> listaProduse = new List<Produs>();

        // Adauga produse in lista.
        listaProduse.Add(new Produs(1000, "pix"));
        listaProduse.Add(new Produs(1001, "minge"));
        listaProduse.Add(new Produs(1002, "caiet"));
        listaProduse.Add(new Produs(1003, "calculator"));
        listaProduse.Add(new Produs(1004, "joc"));
        listaProduse.Add(new Produs(1005, "camasa"));

        // Scriem in consola elementele din lista de produse
        Console.WriteLine();
        foreach (Produs unProdus in listaProduse) {
            Console.WriteLine(unProdus);
        }
        // Verificam daca in listaProduse se afla produsul cu ID-ul #1003.
        //Metoda Contains apeleaza metoda IEquitable.Equals din clasa Produse.
        Console.WriteLine("\nLista contine produsul cu ID-ul (\"1003\"): {0}",
        listaProduse.Contains(new Produs(1003, "")));
```

```
// Inseram un item nou la pozitia 2.
        Console.WriteLine("\nInseram(2, \"1015\")");
        listaProduse.Insert(2, new Produs(1015, "masina"));
        foreach (Produs unProdus in listaProduse)  {
            Console.WriteLine(unProdus);
        }
//Accesam un element din colectie printr-un index
        Console.WriteLine("\nlistaProduse[3]: {0}", listaProduse[3]);
        Console.WriteLine("\nStergem(\"1003\")");
  // Stergem produsul cu ID-ul 1003 chiar daca numeProdus este diferit.
  // Acest lucru este posibil deoarece metoda Equals verifica doar ID-ul produselor.
        listaProduse.Remove(new Produs(1003, "fructe"));
        Console.WriteLine();
        foreach (Produs unProdus in listaProduse) {
            Console.WriteLine(unProdus);
        }

        Console.WriteLine("\nRemoveAt(3)");
   // Aceasta va sterge produsul cu indexul 3.
        listaProduse.RemoveAt(3);
        Console.WriteLine();
        foreach (Produs unProdus in listaProduse )  {
            Console.WriteLine(unProdus);
        }
    }
}
```

Rezultat:

Produs_ID: 1000   Nume: pix
Produs_ID: 1001   Nume: minge
Produs_ID: 1002   Nume: caiet
Produs_ID: 1003   Nume: calculator
Produs_ID: 1004   Nume: joc
Produs_ID: 1005   Nume: camasa
Lista contine produsul cu ID-ul ("1003"): True
Inseram(2, "1015")
Produs_ID: 1000   Nume: pix
Produs_ID: 1001   Nume: minge
Produs_ID: 1015   Nume: masina
Produs_ID: 1002   Nume: caiet
Produs_ID: 1003   Nume: calculator
Produs_ID: 1004   Nume: joc
Produs_ID: 1005   Nume: camasa
listaProdus[3]: Produs_ID: 1002   Nume: caiet
Stergem("1003")

Produs_ID: 1000   Nume: pix
Produs_ID: 1001   Nume: minge
Produs_ID: 1015   Nume: masina
Produs_ID: 1002   Nume: caiet
Produs_ID: 1004   Nume: joc
Produs_ID: 1005   Nume: camasa
RemoveAt(3)

Produs_ID: 1000   Nume: pix
Produs_ID: 1001   Nume: minge
Produs_ID: 1015   Nume: masina
Produs_ID: 1004   Nume: joc
Produs_ID: 1005   Nume: camasa

```
//Comparati prezentul program cu un program similar prezentat in cursul 9. Notati diferenta dinte interfetele
//IComparable si IComparer, pe de o parte, si  interfetele generice IComparable<T> si IComparer<T>, pe de alta parte.
//Mai precis, daca metodele CompareTo() si Compare() ale interfetelor IComparable si respectiv IComparer
// au ca parametrii obiecte de tip Object, pentru CompareTo() si Compare(), ale interfetelor IComparable<T> si
//IComparer<T>, parametrii sunt de tipul T. In consecinta, NU trebuie sa ne asiguram, asa cum am facut in programul
//din cursul 9, ca parametrii sunt de tipul dorit. Genericele au asa numita facilitate type-safe.
using System;
using System.Collections.Generic;

public class Produs: IComparable<Produs> {
    public int produs_ID;
    public string numeProdus;

    public Produs(int produs_ID, string numeProdus) {
        this.produs_ID = produs_ID;
        this.numeProdus = numeProdus;
    }
    public override string ToString() {
        return "Produs_ID: " + produs_ID + "   Nume: " + numeProdus;
    }
    //Metoda CompareTo(T) din IComparable<T> are ca parametru formal tipul T.
    //Metoda CompareTo(object) din IComparable are ca parametru formal tipul object.
    public int CompareTo(Produs prod) {
        return this.produs_ID - prod.produs_ID;
    }
}
```

```
public class ComparaNumeProduse : IComparer<Produs> {
   public static IComparer<Produs> Default = new ComparaNumeProduse();

//Metoda Compare(T,T) din IComparable<T> are ca parametrii formali tipul T.
//Metoda CompareTo(object, object) din IComparable are ca parametrii formali tipul object.
   public int Compare(Produs p1, Produs p2) {
//Apelam proprietatea Default a clasei Comparer<string> pentru a returna un obiect al acestei
clase.  Prin intermediul acestui obiect utilizam implementarea standard a metodei
Compare(string,string)
      return Comparer<string>.Default.Compare(p1.numeProdus, p2.numeProdus);
   }
}

class Exemplu{
   public static void Main() {
      List<Produs> listaProduse = new List<Produs>();
// Adaugam produse in lista.
      listaProduse.Add(new Produs(1007, "pix"));
      listaProduse.Add(new Produs(1020, "minge"));
      listaProduse.Add(new Produs(1050, "caiet"));
      listaProduse.Add(new Produs(1003, "calculator"));
      listaProduse.Add(new Produs(1000, "joc"));
      listaProduse.Add(new Produs(1005, "camasa"));
```

```
// Scriem elementele din lista de produse (nesortate)
        Console.WriteLine("Elementele colectiei sunt:");
        foreach (Produs unProdus in listaProduse) {
            Console.WriteLine(unProdus);
        }

//Sortam lista cu comparatorul default (dupa ID).
//Metoda Sort() apeleaza automat CompareTo(T).
//In cazul de fata, Sort() apeleaza CompareTo(Produs) implementata de clasa Produs.
        Console.WriteLine();
        Console.WriteLine("Produse sortate cu comparatorul default (dupa ID):");
        listaProduse.Sort();
// Scriem elementele din lista de produse, sortate dupa ID.
        foreach (Produs unProdus in listaProduse) {
            Console.WriteLine(unProdus);
        }
```

```
//Sortam lista (dupa nume) prin intermediul metodei Sort(IComparer<T>). Aceasta apeleaza
//automat metoda Compare(T,T).
//In cazul de fata, Sort(IComparer<Produs>) apeleaza Compare(Produs, Produs)
//implementata de clasa ComparaNumeProduse.
//Aceasta clasa, creaza un obiect care este referit prin intermediul referintei Default de tip
//interfata IComparer<Produs>.

        Console.WriteLine();
        Console.WriteLine("Produse sortate cu comparatorul non-default (dupa nume):");
        listaProduse.Sort(ComparaNumeProduse.Default);
        // Scriem elementele din lista de produse, sortate dupa nume.
        foreach (Produs unProdus in listaProduse) {
            Console.WriteLine(unProdus);
        }

    }
}
```

Rezultat:
Elementele colectiei sunt:
Produs_ID: 1007 Nume: pix
Produs_ID: 1020 Nume: minge
Produs_ID: 1050 Nume: caiet
Produs_ID: 1003 Nume: calculator
Produs_ID: 1000 Nume: joc
Produs_ID: 1005 Nume: camasa

Produse sortate cu comparatorul default (dupa ID):
Produs_ID: 1000 Nume: joc
Produs_ID: 1003 Nume: calculator
Produs_ID: 1005 Nume: camasa
Produs_ID: 1007 Nume: pix
Produs_ID: 1020 Nume: minge
Produs_ID: 1050 Nume: caiet

Produse sortate cu comparatorul non-default (dupa nume):
Produs_ID: 1050 Nume: caiet
Produs_ID: 1003 Nume: calculator
Produs_ID: 1005 Nume: camasa
Produs_ID: 1000 Nume: joc
Produs_ID: 1020 Nume: minge
Produs_ID: 1007 Nume: pix

Clasa Dictionary<TKey, TValue>

Reprezintă o colecție de chei și valori.

Proprietățile clasei Dictionary<TKey, TValue>

Comparer	Returnează (obține) IEqualityComparer<T> care este folosită pentru a determina egalitatea cheilor din Dictionary<TKey,TValue>.
Count	Returnează numărul de perechi cheie/valoare cuprinse în Dictionary <TKey, TValue>.
Keys	Returnează o colecție (Dictionary <TKey, TValue>.KeyCollection) care conține cheile din Dictionary<TKey, TValue>.
Values	Returnează o colecție (Dictionary<TKey,TValue>.ValueCollection) care conține valorile din Dictionary<TKey, TValue>.
this[TKey]	Returnează sau setează valoarea asociată cheiei specificate (indexarea clasei List<T>).

Metodele clasei Dictionary<TKey, TValue>

Add (TKey, TValue)	Adaugă cheia specificată și valoarea în dicționar.
Clear	Elimină toate cheile și valorile din Dictionary <TKey, TValue>.
ContainsKey(TKey)	Determină dacă Dictionary<TKey, TValue> conține cheia specificată.
ContainsValue(TValue)	Determină dacă Dictionary <TKey, TValue> conține o anumită valoare.
GetEnumerator	Returnează un enumerator (structura Dictionary<TKey, TValue>.Enumerator) care iterează dictionarul Dictionary<TKey, TValue> . (Metoda MoveNext() din structura Dictionary<TKey, TValue>.Enumerator este apelata automat de bucla foreach)
GetObjectData	Implementeaza interfata System.Runtime.Serialization.ISerializable și returnează datele necesare pentru a serializa o instanță Dictionary<TKey, TValue> .
OnDeserialization	Implementeaza interfataSystem.Runtime.Serialization.ISerializable și produce un eveniment când deserializarea este completă.
Remove(TKey)	Șterge valoarea, corespunzătoare cheiei specificată, din Dictionary <TKey, TValue> .
TryGetValue (TKey, out TValue)	Returnează valoarea asociată cheiei specificată.

Următorul program demonstrează modul în care se poate utiliza clasa Dictionary<TKey, TValue> împreună cu proprietățile și metodele sale, însă și alte clase precum:
Dictionary <TKey, TValue>.KeyCollection,
Dictionary <TKey, TValue>.ValueCollection,
KeyNotFoundException
sau structuri:
KeyValuePair<TKey, TValue > .

Exemplu: Dictionary<TKey, TValue>

```
using System;
using System.Collections.Generic;

public class Produs  {
    public int pretProdus;
    public string numeProdus;

    public Produs(int pretProdus, string numeProdus)  {
        this.pretProdus = pretProdus;
        this.numeProdus = numeProdus;
    }

    public override string ToString() {
        return  numeProdus +" la pretul de " + pretProdus + " lei" ;
    }
}
```

```
class Exemplu  {
    public static void Main(){
        // Cream un dictionar (Dictionary) de Produse avand drept cuvinte cheie stringuri
        Dictionary<string, Produs> listaProduse =  new Dictionary<string, Produs>();

        //Cream cateva produse si le adaugam in catalog.
        //Cuvintele cheie nu pot fi duplicate. Valorile insa se pot duplica.
        Produs pix = new Produs(5, "Pix");
        listaProduse.Add("#1000", new Produs(50, "Minge"));
        listaProduse.Add("#1001", pix);
        listaProduse.Add("#1002", pix);
        listaProduse.Add("#1003", new Produs(100, "Carte"));

        // Metoda Add lanseaza o exceptie daca in lista se afla deja cuvantul cheie.
        try {
            listaProduse.Add("#1003", new Produs(1000, "Telefon"));
        }
        catch (ArgumentException) {
            Console.WriteLine("Elementul avand cuvantul cheie = \"#1003\" este deja in lista.");
        }
```

```
// Un element, poate fi accesat prin intermediul unei indexari.
        Console.WriteLine("Corespunzator ID-ului = #1003, se afla produsul: {0}.",
           listaProduse["#1003"]);

 // Indexarea poate fi utilizata pentru a schimba valoarea asociata unui cuvant cheie.
        listaProduse["#1003"] = pix;
        Console.WriteLine("Corespunzator ID-ului = #1003, se afla produsul: {0}.",
           listaProduse["#1003"]);

// Daca cuvantul cheie nu se afla in lista, atunci indexatea adauga acel element in lista.
        listaProduse["#1004"] = new Produs(3, "Un kg mere");

 // Indexarea lanseaza o exceptie daca cheia nu este in lista.
        try {
            Console.WriteLine("Corespunzator ID-ului = #1005, se afla produsul: {0}.",
           listaProduse["#1005"]);
        }
        catch (KeyNotFoundException) {
            Console.WriteLine("Cheia = #1005 nu a fost gasita.");
        }
```

```
// Daca un program are de incercat o serie de chei care ar putea sa nu fie in lista, atunci
// se poate utiliza proprietatea TryGetValue ca mai jos.
        Produs valoare;
        if (listaProduse.TryGetValue("#1004", out valoare)) {
            Console.WriteLine("Corespunzator ID-ului = #1004, se afla produsul: {0}.",
           valoare);
        }
        else {
            Console.WriteLine("Cheia = \"#1004\" nu a fost gasita.");
        }

  // Se poate utiliza metoda ContainsKey() pentru a vedea daca exista sau nu o anumita cheie.
        if (!listaProduse.ContainsKey("#1005")) {
            listaProduse.Add("#1005", new Produs(2, "Paine"));
            Console.WriteLine("Corespunzator ID-ului = #1005, se afla produsul: {0}.",
           listaProduse["#1005"]);
        }

 // Bucla foreach returneaza obiecte de tipul KeyValuePair, o structura din acelasi
 // spatiu de nume System.Collection.Generic. Acelasi concept privind foreach a fost intalnit si in
//cazul colectiilor non-generice (vezi clasa DictionaryBase si structura DictionaryEntry).
        Console.WriteLine();
        foreach (KeyValuePair<string,Produs> kvp in listaProduse) {
           Console.WriteLine("Corespunzator ID-ului = {0}, se afla produsul: {1}.",
           kvp.Key, kvp.Value);
        }
```

```
// Pentru a obtine doar valorile elementelor din lista se poate utiliza proprietatea Values
//care returneaza un obiect de tipul clasei Dictionary<string, Produs>.ValueCollection
    Dictionary<string, Produs>.ValueCollection valorileColectiei=listaProduse.Values;

 // Elementele colectiei ValueCollection sunt puternic tipizate,
 // in cazul nostru acestea sunt de tipul Produs.
    Console.WriteLine("Elementele colectiei
            Dictionary<string, Produs>.ValueCollection sunt de tipul Produs:");
      foreach (Produs p in valorileColectiei) {
         Console.WriteLine("{0}", p);
      }

  // Pentru a obtine cheile elementelor din lista se poate utiliza proprietatea Keys care
  / returneaza un obiect de tipul clasei Dictionary<string, Produs>.KeyCollection
       Dictionary<string, Produs>.KeyCollection cheileColectiei =listaProduse.Keys;
```

```
// Elementele colectiei KeyCollection sunt puternic tipizate,
// in cazul nostru acestea sunt de tipul string.
       Console.WriteLine("Elementele colectiei
               Dictionary<string, Produs>.KeyCollection sunt de tipul string:");
       foreach (string s in cheileColectiei) {
           Console.WriteLine("{0}", s);
       }


 // Se poate utiliza metoda Remove() pentru a sterge o pereche cheie/valoare din
 //colectie.
       Console.WriteLine("\nStergem(#1003)");
       listaProduse.Remove("#1003");
       if (!listaProduse.ContainsKey("#1003")) {
           Console.WriteLine("Cheia #1003 nu a fost gasita.");
       }
    }
}
```

Rezultat:

Elementul avand cuvantul cheie = "#1003" este deja in lista.
Corespunzator ID-ului = #1003, se afla produsul: Carte la pretul de 100 lei.
Corespunzator ID-ului = #1003, se afla produsul: Pix la pretul de 5 lei.
Cheia = #1005 nu a fost gasita.
Corespunzator ID-ului = #1004, se afla produsul: Un kg mere la pretul de 3 lei.
Corespunzator ID-ului = #1005, se afla produsul: Paine la pretul de 2 lei.

Corespunzator ID-ului = #1000, se afla produsul: Minge la pretul de 50 lei.
Corespunzator ID-ului = #1001, se afla produsul: Pix la pretul de 5 lei.
Corespunzator ID-ului = #1002, se afla produsul: Pix la pretul de 5 lei.
Corespunzator ID-ului = #1003, se afla produsul: Pix la pretul de 5 lei.
Corespunzator ID-ului = #1004, se afla produsul: Un kg mere la pretul de 3 lei.
Corespunzator ID-ului = #1005, se afla produsul: Paine la pretul de 2 lei.
Elementele colectiei Dictionary<string, Produs>.ValueCollection sunt de tipul Produs:
Minge la pretul de 50 lei
Pix la pretul de 5 lei
Pix la pretul de 5 lei
Pix la pretul de 5 lei
Un kg mere la pretul de 3 lei
Paine la pretul de 2 lei
Elementele colectiei Dictionary<string, Produs>.KeyCollection sunt de tipul string:
#1000
#1001
#1002
#1003
#1004
#1005
Stergem(#1003)
Cheia #1003 nu a fost gasita.

Colecții sortate

Clasa negenerica System.Collections.SortedList, clasa generică System.Collections.Generic.SortedList <TKey, TValue> și clasa generică System.Collections.Generic.SortedDictionary <TKey, TValue> sunt similare clasei Dictionary < TKey, TValue > , în sensul că toate implementează interfața IDictionary, dar își păstrează elementele sortate după o cheie.

Cele trei clase au mai multe caracteristici în comun:
-Toate cele trei clase implementează System.Collections.IDictionary. Cele doua clase generice implementează, de asemenea, interfeța generică System.Collections.Generic.IDictionary <TKey, TValue> .

-Fiecare element este o pereche cheie valoare (cu scopul de a putea fi enumerate). Cand se utilizeaza bucla foreach, SortedList returnează obiecte DictionaryEntry, în timp ce cele două tipuri generice returnează obiecte KeyValuePair <TKey, TValue> .

-Elementele sunt sortate conform implementării interfeței System.Collections.IComparer, pentru clasa negenerică SortedList, și respectiv implementării interfeței System.Collections.Generic.IComparer <T>, pentru cele două clase generice.

-Fiecare clasă oferă proprietăți (proprietățile Keys și Values) care returnează colecții care conțin numai cheile sau doar valorile.

Însă sunt și o serie de aspecte care diferențiază clasa SortedDictionary <TKey, TValue> față de clasele SortedList și SortedList <TKey, TValue> . Spre exemplu:

a) proprietățile Keys și Values din SortedList și SortedList <TKey, TValue> returnează chei și valori care sunt indexate. Așadar se poate utiliza o indexare. În schimb, pentru SortedDictionary <TKey, TValue> nu putem proceda la fel. (Vezi exemplul urmator).

b) SortedList și SortedList <TKey, TValue> utilizează mai puțină memorie decât SortedDictionary <TKey, TValue>.

Exemplu: SortedList<TKey, TValue> si SortedDictionary<TKey, TValue>

```
using System;
using System.Collections.Generic;

public class Produs{
    public int pretProdus;
    public string numeProdus;

    public Produs(int pretProdus, string numeProdus)    {
        this.pretProdus = pretProdus;
        this.numeProdus = numeProdus;
    }

    public override string ToString() {
        return  numeProdus + " la pretul de " + pretProdus+ " lei" ;
    }
}
```

```
class Exemplu{
    public static void Main(){
        SortedList<int, Produs> sortedListProduse=new SortedList<int,
Produs>();

        // Adaugam produse in lista SortedList<int, Produs>.
        sortedListProduse.Add(703, new Produs(2, "Pix"));
        sortedListProduse.Add(345,new Produs(50, "Minge"));
        sortedListProduse.Add(450,new Produs(2, "Caiet"));
        sortedListProduse.Add(606,new Produs(1000, "Telefon"));


        // Scriem elementele din lista SortedList<int, Produs>.
        Console.WriteLine("Elementele colectiei SortedList<int, Produs> sunt:");
        foreach (KeyValuePair<int,Produs> kvp in sortedListProduse)
        {
            Console.WriteLine("Cheia={0}, Valoarea={1}",kvp.Key, kvp.Value);
        }
        Console.WriteLine();
```

```
SortedDictionary<int, Produs> sortedDictionaryProduse = new SortedDictionary<int,
        Produs>();

    // Adaugam produse in lista sortedDictionaryProduse<int, Produs>.
    sortedDictionaryProduse.Add(703, new Produs(2, "Pix"));
    sortedDictionaryProduse.Add(345, new Produs(50, "Minge"));
    sortedDictionaryProduse.Add(450, new Produs(2, "Caiet"));
    sortedDictionaryProduse.Add(606, new Produs(1000, "Telefon"));


    // Scriem elementele din lista sortedDictionaryProduse<int, Produs>.
    Console.WriteLine("Elementele colectiei sortedDictionaryProduse<int, Produs>
        sunt:");
    foreach (KeyValuePair<int, Produs> kvp in sortedDictionaryProduse)
    {
      Console.WriteLine("Cheia={0}, Valoarea={1}", kvp.Key, kvp.Value);
    }
    Console.WriteLine();
```

```
//Intrucat propretatea Values (sau Keys) din SortedList<TKey, TValue>  intoarce un
//IList<T>, iar aceasta clasa contine
//indexarea this[int], putem accesa elementele listei IList<T> printr-un index.
        IList<Produs> listaProduse = sortedListProduse.Values;
        Console.WriteLine("Elementul cu indexul 2 in listaProduse este: {0}",
                listaProduse[2]);

//In schimb, pentru SortedDictionary<TKey, TValue> rezultatul intors de proprietatea
//Values este SortedDictionary<TKey, TValue>.ValueCollection, iar aceasta clasa nu
//este indexata.
//NU putem proceda ca mai sus. Daca stergeti comentariul de mai jos se obtine eroare.
        SortedDictionary<int, Produs>.ValueCollection valoarecolectieDictionary =
                                                    sortedDictionaryProduse.Values;
       // Console.WriteLine(valoarecolectieDictionary[2]);

    }
}
```

Rezultat:

Elementele colectiei SortedList<int, Produs> sunt:
Cheia=345, Valoarea=Minge la pretul de 50 lei
Cheia=450, Valoarea=Caiet la pretul de 2 lei
Cheia=606, Valoarea=Telefon la pretul de 1000 lei
Cheia=703, Valoarea=Pix la pretul de 2 lei

Elementele colectiei sortedDictionaryProduse<int, Produs> sunt:
Cheia=345, Valoarea=Minge la pretul de 50 lei
Cheia=450, Valoarea=Caiet la pretul de 2 lei
Cheia=606, Valoarea=Telefon la pretul de 1000 lei
Cheia=703, Valoarea=Pix la pretul de 2 lei

Elementul cu indexul 2 in listaProduse este: Telefon la pretul de 1000 lei

Cozi și stive

System.Collections.Queue, System.Collections.Generic.Queue <T> și System.Collections.Concurrent.ConcurrentQueue <T> sunt clase colecții first-in, first- out care implementeaza interfata ICollection. Clasele generice implementează și interfața ICollection < T >.

System.Collections.Stack, System.Collections.Generic.Stack <T> și System.Collections.Concurrent.ConcurrentStack <T> sunt clase colecții last-in, first- out care implementeaza interfata ICollection. Clasele generice implementează și interfața ICollection < T >.

Cozile și stive sunt utile atunci când aveți nevoie de depozitare temporară a unor informații, adică atunci când doriți să renunțați la un element după preluarea valorii sale.
Utilizați o coadă dacă aveți nevoie să accesați informațiile în aceeași ordine în care acestea sunt stocate în colecție.
Utilizați o stivă dacă aveți nevoie să accesați informațiile în ordine inversă.

Trei operații principale pot fi efectuate asupra unei cozi și elementelor sale:

- Enqueue — Adaugă un element la sfârșitul cozii.
- Dequeue — Elimină cel mai vechi element de la începutul cozii. Metoda TryDequeue returneaza false în cazul în care valoarea nu poate fi eliminată.
- Peek — Returnează cel mai vechi element care este la începutul cozii, dar nu-l scoate din coadă.

Trei operații principale pot fi efectuate asupra unei stive și elementelor sale:

- Push — Introduce un element în partea de sus a stivei.
- Pop — Inlătură un element din partea de sus a stivei.
- Peek — Returnează un element care este în partea de sus a stivei, dar nu-l scoate din stivă.